Flexible Control and Interprocess Communication on the Rogue GPS Receiver

by

Robert Blau

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology

May 21, 1999

Signature of Author _____
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by _____
Tomas Lozano-Perez
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

FLEXIBLE CONTROL AND INTERPROCESS COMMUNICATION ON THE ROGUE GPS RECEIVER

by

Robert Blau

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 1999

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The Rogue receivers are a series of custom high-accuracy Global Positioning System receivers being developed at NASA's Jet Propulsion Laboratory. This thesis describes two additions to the RogueOS, a custom operation system developed for these receivers. The first addition is an implementation of interprocess communication. This provides separate processes running on the receiver the power to pass arbitrary data structures easily and quickly to each other. The second addition is a flexible control system that allows various functional units to be linked together in a highly adaptive way. The intent of the control system is to make robust real-time, embedded, and intelligent seeming applications easy to develop. This control system is the first step in making the Rogue receivers autonomous entities capable of carrying out complex missions in low earth orbit satellites without the need for human intervention.

Thesis Supervisor: Tomas Lozano-Perez
Title: Professor of Computer Science and Electrical Engineering

ACKNOWLEDGMENTS

I would like to thank my family and friends for believing in me with an almost religious fervor. Without their encouragement I would never have had the courage to set my goals so high and without their support I would never have had the will to achieve them.

I would like to thank the Rogue group at JPL, who had the patience to take in a clueless intern and provide him with the information and tools that made this thesis a reality. I would like to thank Charlie Dunn in particular, for the discussions we had about the Cloud, and for his guidance with understanding the RogueOS.

Finally, I want to give heartfelt thanks to all the professors at M.I.T. whose teachings have given me the fuel that will feed inspiration for the rest of my life. I would like to especially thank Prof. Lozano-Perez for being the voice of reason (a.k.a. advisor) when I needed it.

Table of Contents

List of Figures 7

Chapter 1. Introduction 8

1.1 The Global Positioning System 8
1.2 The Rogue Receivers 9
1.3 The Rogue Operating System (RogueOS) 10

Chapter 2. The Communications Library 12

2.1 The Original Implementation 12
2.2 The New Implementation 13

Chapter 3. Background for the Control System 16

3.1 Autonomous Spacecraft 16
3.2 Robotic Control Systems 17
3.3 What creatures should be able to do 18
3.4 How creatures have worked 19
3.4.1 Symbolic Systems 19
3.4.2 Behavioral Systems 20
3.5 How creatures should do work 22
3.5.1 Nature 22
3.5.2 Society of Mind 22
3.5.3 Streams and Counter-Streams 23
3.5.4 Routines 24
3.5.6 One-Shot Learning 26
3.6 Summary 27

Chapter 4. The Control System Design 28

4.1 An outline of the design 28
4.2 The data layer 29
4.2.1 Data behaviors 29
4.2.2 Sensors and Sensor behaviors 30
4.2.3 Connections between data behaviors 30
4.2.4 Connections to the action layer 31
4.2.5 Data behaviors and goals 31
4.2.6 More on bottom-up activation 32
4.2.7 Parallel execution 32
4.2.8 Deactivation 33
4.3 The action layer 34
4.3.1 Actuator interfaces 34
4.3.2 Action behaviors 34
4.4 The goal engine 35
4.5 Example Networks 35
4.5.1 A walking network 36
4.5.2 Genghis' walk 38
4.5.3 A simplified GPS solution 41

Chapter 5. Contributions 43

5.1 The communications library 43
5.2 DAG Networks compared to symbolic systems and subsumption 43
5.3 Do DAG Networks meet the criteria? 45

5.3 FUTURE WORK 45
A.1 DISTREAM.H 47
A.2 DISTREAM.TMPL.H 48
A.3 DOSTREAM.H 49
A.4 DOSTREAM.TMPL.H 49
A.5 INSTANTIATIONS.CP 50
A.6 SHAREDSOCKET.H 51
A.7 SHAREDSOCKET.TMPL.H 52
A.8 STREAMREGISTRY.H 54
A.9 STREAMREGISTRY.TMPL.H 54

Appendix B. Communications Demo 57

B.1 SCDEMORXMAIN.CP 57
B.2 SCDEMORXTHREAD.CP 57
B.3 SCDEMORXTXMAIN.CP 58
B.4 SCDEMOTXMAIN.CP 58
B.5 SCDEMOTXTHREAD.CP 58
B.6 SCDEMORXTHREAD.H 59
B.7 SCDEMORXTHREAD.H 59

Appendix C. Commlib Differences 61

Bibliography 63

List of Figures

FIGURE 1.1 – TYPICAL GPS ERROR IN METERS (PER SATELLITE) [TRIMBLE, 1996] 9
FIGURE 1.2 – TYPICAL GPS POSITION ACCURACY [TRIMBLE, 1996] 9
FIGURE 1.3 – A ROGUE GPS RECEIVER 9
FIGURE 1.4 – MEMORY PROTECTION IN ROGUE OS [DUNN, 1998] 10
FIGURE 1.5 – TASK SCHEDULING IN THE ROGUEOS [DUNN, 1998] 11

FIGURE 2.1 – ORIGINAL ROGUEOS COMMUNICATIONS [DUNN, 1997] 12
FIGURE 2.2 – THE REDESIGNED COMMUNICATIONS LIBRARY 14

FIGURE 3.1 – THE NMRA ARCHITECTURE [SMITH, 1997] 16
FIGURE 3.2 – GENGHIS, ONE OF THE FIRST CREATURES 18
FIGURE 3.3 – SHAKEY 19
FIGURE 3.4 – WALTER'S TORTOISE 20
FIGURE 3.5 – BEHAVIORAL TORTOISE 24
FIGURE 3.6 – BEHAVIORAL DOES SYMBOLIC 25

FIGURE 4.1 – AN EXAMPLE DAG NETWORK 28
FIGURE 4.2 – WALKING 29
FIGURE 4.3 – DEADLOCK 33
FIGURE 4.4 – ASYNCH ONOUS WALK BEHAVIOR ..... R 36
FIGURE 4.5A – RIGHT TEP BEHAVIOR ..... S 36
FIGURE 4.5B – LEFT STEP BEHAVIOR 36
FIGURE 4.6 – A SIMPLE WALKING NETWORK 37
FIGURE 4.7 – STAND DATA BEHAVIOR 38
FIGURE 4.8 – LEG DOWN SENSOR BEHAVIOR 38
FIGURE 4.9 – ALPHA BALANCE BEHAVIOR 38
FIGURE 4.10 – ALPHA ADVANCE BEHAVIOR 40
FIGURE 4.11 – UP LEG TRIGGER 40
FIGURE 4.12 – PATTERN GENERATOR FOR AN ALTERNATING TRIPOD GAIT 40
FIGURE 4.13 – GPS DAG NETWORK 41

Chapter 1. Introduction

This thesis is made up of three parts that describe two additions to the operating system for the Rogue family of Global Positioning System (GPS) receivers. The first part provides the background and context for both of these additions. It provides an overview of the GPS, the receivers, and the operating system. The next part describes the design and implementation of a communications library for the receivers. The last part describes the design and implementation of a control system for the operating system that is to serve as a substrate which programmers can use to easily implement robust reactive systems.

1.1 The Global Positioning System

GPS was originally a military technology. Near the end of the cold war the targeting systems on intercontinental ballistic missiles became so accurate that they could target enemy's missile silos reliably. The ability to destroy your opponent's missiles had a great effect on the balance of power. In order to attain such accuracy you had to know the missile's precise launch location. When you launch the missiles from land, as the Soviet Union did, this is an easy problem, but the bulk of the U.S. nuclear arsenal was at sea on submarines. In order to match the Soviet accuracy the U.S. had to come up with a way to allow these submarines to surface and quickly fix their exact position anywhere in the world. This is the problem that GPS solved [Trimble, 1996].

GPS consists of three segments: a space segment, a control segment, and a user segment. The space segment consists of a constellation of 24 satellites orbiting the earth at 10,900 nautical miles. The constellation is designed such that between five and eight satellites are visible at any time from any point on the earth [Dana, 1994]. These satellites broadcast signals to the earth containing ~~positional~~ and timing data. Using the signals from four satellites a receiver can use resection to calculate both its position in three coordinates to sub-meter accuracy and universal time to atomic accuracy.

The control segment, or ground segment consists of five monitoring stations that monitor the health of all the GPS satellites. These stations are located in Hawaii, Ascension Island, Diego Garcia, Kwajalein, and Colorado Springs. These stations collect exact position data for each satellite and relay the information to the master control station in Colorado Springs. The master station translates this information into corrections, which it then sends to the satellites to be incorporated into the data sent to the user segment.

Error Source	Standard GPS	Differential GPS
Satellite Clocks	1.5	0
Orbit Errors	2.5	0
Ionosphere	5.0	0.4
Troposphere	0.5	0.2
Receiver Noise	0.3	0.3
Multipath	0.6	0.6
SA	30	0

Figure 1.1 - Typical GPS Error in Meters (per satellite) [Trimble, 1996]

	Standard GPS	Differential GPS
Horizontal	50	1.3
Vertical	78	2.0
3-D	93	2.8

Figure 1.2 - Typical GPS Position Accuracy [Trimble, 1996]

The user segments consists of all GPS receivers. These receivers locate GPS satellite signals, decode the signals, then use the information in the signals to calculate latitude, longitude, altitude, and universal time. The basic calculations result in positions accurate to within 100 meters, while more advanced calculation can yield sub-meter accuracy (Figure 1.1 and Figure 1.2).

→ we do much beter

1.2 The Rogue Receivers

The Rogue Family of GPS receivers is a set of extremely configurable high accuracy GPS receivers intended to satisfy NASA's GPS needs. The purpose of these receivers is to act as scientific instruments, returning atomic clock accuracy time and accurate ephemeris (positional data). Recently many low earth orbit missions have been proposed that would use such information to recover information about the Earth. The GPS On A Chip (GOAC) series of Rogue receivers answers the needs of these missions [Franklin, 1998].

The basic GOAC receivers use the PowerPC 603e microprocessor and run at over 150 MHz. The hardware allows for multiple configurations and even expansion in the GPS itself. Custom ASICs (application specific integrated circuits) handle the signal processing aspects of computer GPS solutions, leaving the CPU free to handle higher level functionality. The GOAC receivers allow for up to four ASICs, each with the ability to track multiple satellites at once. All this allows for a great amount of flexibility in the potential uses of the receiver. All this potential requires a flexible, powerful, and easy to use software base. A custom operating system fills this need.

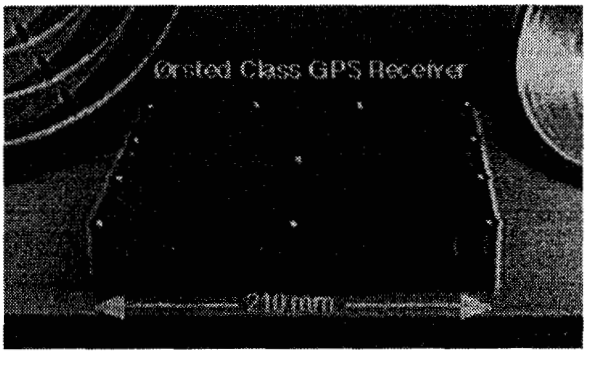

Figure 1.3 - A Rogue GPS Receiver

1.3 The Rogue Operating System (RogueOS)

The RogueOS is a custom operating system developed for the GOAC GPS receivers by the Rogue group at NASA's Jet Propulsion Laboratory (JPL). The operating system runs Apple PEF (PowerPC Executable Formats) in a preemptive multitasking environment. The system gives user code capabilities such as memory management, multi-threading, and interrupt handling. Extra functionality can be made available via libraries that can be dynamically linked at runtime [Dunn, 1997]. The results of this thesis are two runtime libraries for the operating system. The communication library provides separate processes the ability to pass arbitrary data structures between any number of threads. The second library provides the functionality of the control system described in the third part of this thesis. This section will explain the workings of the two parts of the operating system relevant to the libraries: memory protection and multitasking.

Memory protection in the RogueOS takes the form of virtual memory. The virtual memory system has various mappings from virtual memory spaces into physical memory, each virtual memory space is referred to as a *memory context*. Several threads of execution may share a memory context, or they can execute in separate contexts. Threads from one memory context cannot directly manipulate the memory in another. A group of threads executing in the same memory context are referred to as a *process*. Figure 1.4 illustrates this situation. Communication between processes is made possible by using a shared memory interface; addresses in a particular range will be mapped to the same physical memory addresses regardless of the memory context of the executing thread [Dunn, 1998].

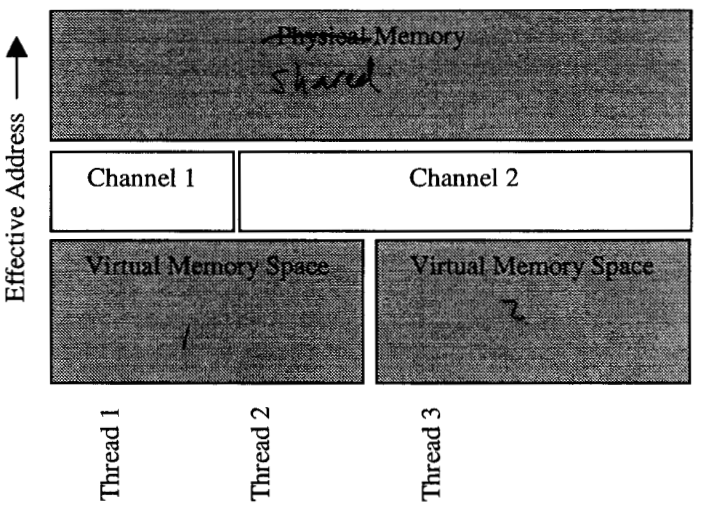

Figure 1.4 - Memory protection in RogueOS [Dunn, 1998]

RogueOS allows for multiple threads to be run simultaneously in a preemptive multitasking environment. A thread manager is responsible for task switching, task prioritization, interrupt dispatching, and handling timed events. A context switch occurs for the following reasons:

- A thread voluntarily gives up the processor
- A higher priority thread wakes up
- A higher priority thread is started
- The time slice for the currently executing thread runs out (a decrementer task switch interrupt is received

The next thread to run upon a context switch is determined in a round robin manner. By default, four tasks from one priority level execute before a task from the next lower level executes once (Figure 1.5). There are five priority levels, from lowest priority to highest they are: Background, Low, Normal, Urgent, and Emergency. Tasks at the Emergency priority level never yield to tasks at lower priority level . s

The operating system has many other features less relevant to this thesis, such as ethernet support, RS-422 support, and power management. Some support features, such as mutexes and locking, will be described as they come up.

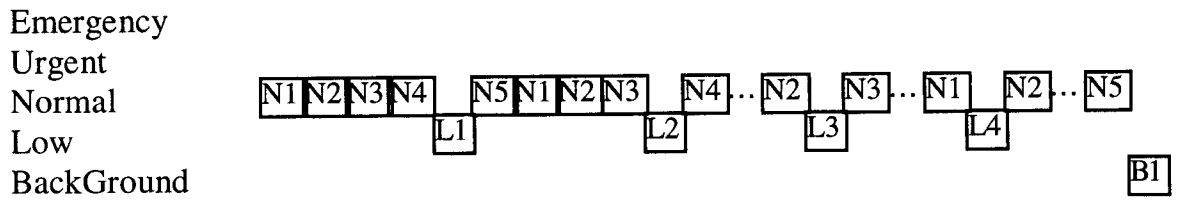

Figure 1.5 - Task scheduling in the RogueOS [Dunn, 1998]

Chapter 2. The Communications Library

This chapter will describe the design and implementation of the communications system in RogueOS. The first section describes the original communications implementation in the operating system, by Charlie Dunn. The second section describes the replacement for this implementation and its capabilities.

2.1 The Original Implementation

The original implementation of the communications library set up a datastream model. The inspiration for this model comes from C++ streams. Once a datastream has been set up, passing objects between two threads is as easy as calling the input and output operators (>> and << respectively). This implementation defined the interface for the next version.

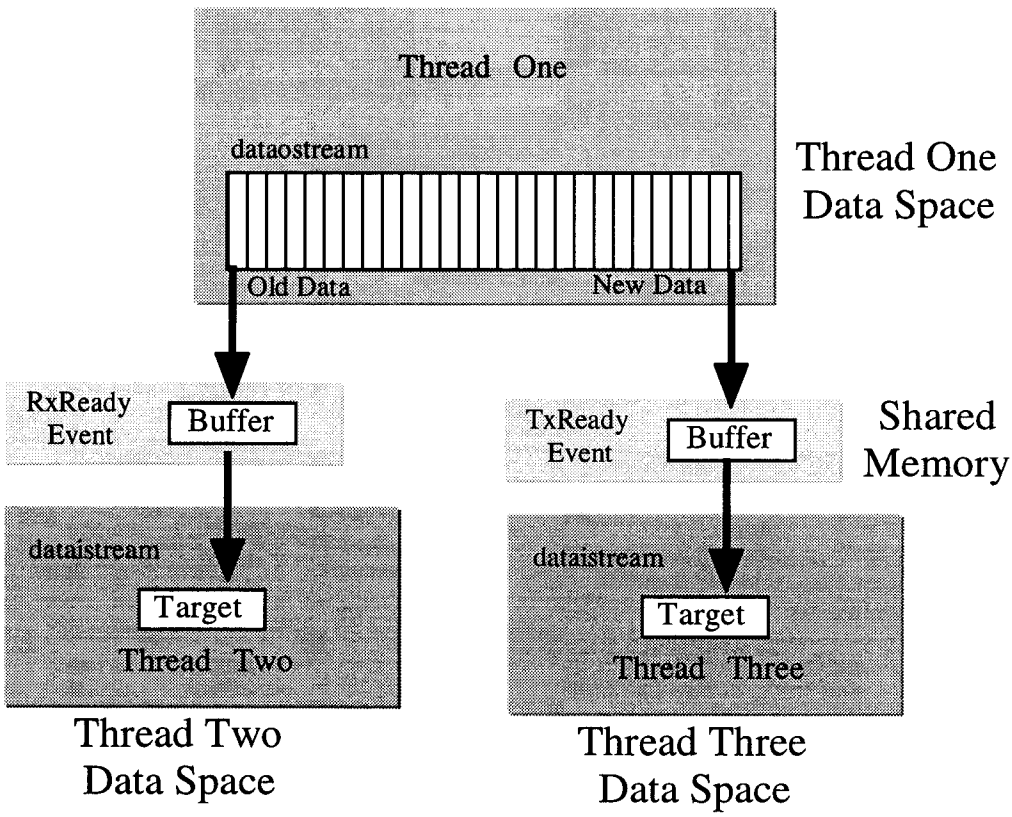

Figure 2.1 - Original RogueOS communications [Dunn, 1997]

Figure 2.1 shows a high level view of this implementation. It is a "file" paradigm – one writer, multiple readers. There is one output stream (dostream) with an associated data buffer. This data buffer exists in the same memory context as the dostream. Input streams (distreams) attach to the dostream, copying data from it upon a read. The data is transferred from one thread to another by events, which copy data from the output memory context to shared memory context and then from there into the input memory context. When all registered readers read past a data element, it is deleted from the data buffer. Data is passed by value, so a specific thread can pass so only one type of data (though polymorphism can be achieved by passing pointers into shared memory). Threads that use dostreams must descend from a pktsource<DataBase> class where DataBase is a base class to all the data types to be transferred. Similarly, threads that use distreams must descend from pktsink<DataBase>. The data types being transferred must define a "Base" type, the same type as the DataBase template argument, passed to pktsource and pktsink.

This implementation had a few drawbacks that needed to be corrected for upcoming missions. First of all, a more general multi-writer multi-reader implementation was needed. Also, due to the data rates the streams needed to handle, a faster implementation was needed. And finally, this implementation had problems working across memory contexts. The event classes would be instantiated with one of the streams, living in either the memory context of the output thread or the memory context of the input thread. This meant that the events could copy objects into shared memory, but would have no way of communicating where those objects were across other memory contexts. The new communications library addresses these problems.

2.2 The New Implementation

The two main ideas behind the re-implementation of the communications library are simplicity and moving the data buffer itself into shared memory. Simplicity is in the design and the interface. Figure 2.2 shows an overview of the design. In this implementation streams have direct connections to the buffer in shared memory.

When a stream is created, it is given an identifying string. This string links the stream with a data buffer in shared memory. All streams created later that are given the same name and pass the same type of data are linked to the same data buffer. Output streams simply place data into this buffer, and input streams simply read it. There are various ways that input streams can behave when they read beyond the end of the buffer. Four modifiers control this behavior; eof_on, eof_off, block, and nonblocking. If a stream is specified block, then the stream will put the thread to sleep when it reads beyond the end of the buffer. If a stream is specified as nonblocking then an exception will be thrown when you read beyond the end of the buffer. If a stream is specified eof_on, then the stream will throw an end of file exception when a reader

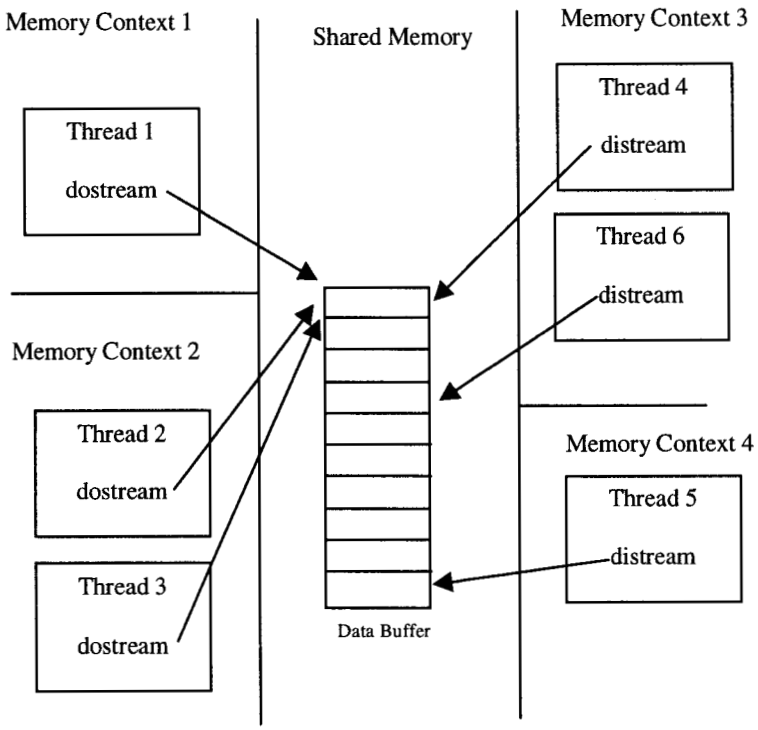

Figure 2.2 - The redesigned communications library

reads beyond the end of the buffer and there are no writers registered. If eof_off is specified, then the stream will just put the thread to sleep, waiting for another writer to connect and send more data.

This implementation gets around the problems with the previous one. First of all there are no special requirements of threads that use streams; they do not have to be descended from any base class. The only requirement made of the data class is that it has a copy constructor, to allow the data to be moved between memory contexts. As the data buffer is now separate from the writer and the reader, it is easy to have multiple readers and writers. The simple implementation means that there is a nice performance increase. The bulk of the time is spent either copying the data or adjusting the size of the data buffer (depending on how complicated the data is). In order to make this work across memory contexts one addition needs to be made.

As it stands, this implementation falls prey to the same fault that the previous implementation succumbed to, there is no way to set us the communication across the shared memory context. The way around this is to compile the library as a system library, so that the code and all variables live in shared memory. A pointer for each data buffer for each data type is instantiated upon the loading of the library. All threads that include the headers for the library are told where the buffer is, thus solving the problem.

This library has been distributed to the members of the Rogue group and has been put to use in the latest mission software. It has proven to work well for all but the most time intensive of

the communications chores. Appendix A lists the source code for the communications library. Appendix B lists the source code for a demonstration of how to use the library. Appendix C is the documentation on the difference between the two versions.

Chapter 3. Background for the Control System

This chapter begins the meat of this thesis: the development and description of the control system architecture for the Rogue receivers (nicknamed the Cloud). The task at hand was to develop a substrate for the rogue programmers to program on top of that offered responsiveness, robustness, and ease of use. This substrate would tie together the work of the various programmers and allow the receiver to act in an autonomous fashion. There are two close analogies to this problem, the development of autonomous spacecraft and the development of autonomous robots. This chapter discusses work done on both of these tasks and draws forth from them capabilities the Cloud should have. It also describes some ideas from Artificial Intelligence that have inspired aspects of the system.

3.1 Autonomous Spacecraft

NASA has recently instituted a New Millennium Program (NMP) whose mission is to achieve a "virtual presence" in space by deploying spacecraft built according to the new NASA "faster, better, cheaper" motto. Spacecraft with some degree of autonomy are needed to fulfill this goal. Autonomous spacecraft will reduce missions operation cost by taking over many of the tasks usually performed by ground stations. They will also improve mission quality by being more reactive to their environment and more failure tolerant than traditional spacecraft. The first NMP missions, the Deep Space program, are designed to showcase new technology being developed at NASA. The first Deep Space mission, Deep Space One (DS1) launched in October 1998. One of the technologies the DS1 mission was to demonstrate was the New Millennium Remote Agent (NMRA), the first Artificial Intelligence system to control an actual spacecraft [Pell, 1996]. Due to delays, the spacecraft was launched without the NMRA, so the software remains untested.

The NMRA is at heart a real-time planning engine. A long-term plan covering the entire mission (*mission profile*) is stored in a part of the system called the *Mission Manager* (MM). Another part of the system, the Executive (EXEC), requests plans from the MM. When this happens the MM selects a set of goals from the mission profile and presents it to the Planner/Scheduler (PS). The PS turns these goals into a plan that the EXEC can run. When the plan is almost all carried out the EXEC sends another request to the MM for another plan [Muscettola, 1997]. Figure 3.1 shows an overview of the NMRA architecture.

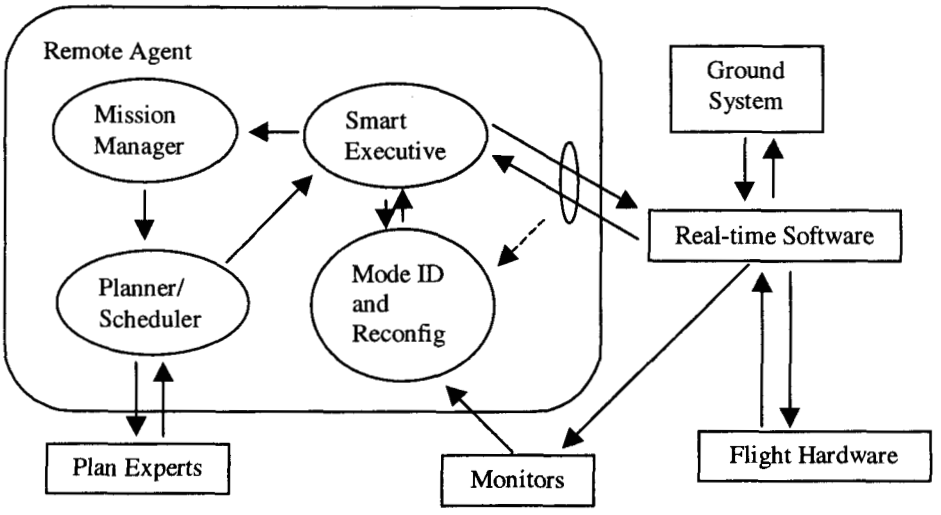

Figure 3.1 - The NMRA Architecture [Smith, 1997]

Plans in the NMRA consist of several parallel timelines each consisting of a sequence of tokens that describe the state of the spacecraft. Tokens have a start time, a stop time, and a duration. Tokens can also have one or more arguments. Constraints determine how different tokens can be arranged.

NMRA suffers from the same limitations that most planning systems suffer from. Interactions between different tokens must be explicitly modeled for the system to know the effects one action has on another. When a complicated system is being developed, the number of these interactions grows exponentially, making explicit modeling difficult. Plans are neither robust nor reactive. Unexpected changes in the environment can throw plans completely off and the spacecraft cannot react to the situation until a new plan has been formulated. These problems are in the areas where the Rogue control system needs performance, indicating a preference away from planning systems.

3.2 Robotic Control Systems

From one point of view, the Rogue receiver is much like a robot. It is able to sense the state of the world and act upon it. Its sensors are antennas and its actuators are communication ports. Using this analogy, work done in autonomous robotics can be used to control the receiver. Unfortunately, robotics has not been able to produce a robot capable of accomplishing complex tasks reliably in a complex environment. The vast majority of robots in the world operate in a simplified world, where unexpected things do not happen. Assembly line robots are not able to

cope with parts arriving in random orientations, mail delivery robots cannot reorient reliably if bumped.

To contrast with these types of robots, Rodney Brooks defines a *Creature* as "a completely autonomous mobile agent that co-exists in the world with humans, and is seen by those humans as an intelligent being in its own right" [Brooks, 1991]. A control system for such a creature bridges the chasm between sensor input and actuator output, allowing the creature to react intelligently to its environment. The goal for the Rogue receivers is to appear as creatures, intelligent, reactive, and robust. The rest of this thesis presents an architecture for such a control system, nicknamed the Cloud, more technically called Data-Action-Goal Networks (DAG Networks).

The rest of this chapter is an overview of robotic control systems. The next part describes what capabilities we should expect these systems to have. After that is a brief discussion of the history of these systems and an evaluation of what approaches achieve which capabilities. The final part of this chapter discusses characteristics DAG Nets should have, drawing inspiration from powerful ideas in modern Artificial Intelligence.

3.3 What creatures should be able to do

To deserve the title, a creature should display a certain level of functionality. The definition of creature given above points the way to this baseline functionality. First, a creature should be autonomous. An autonomous creature is one that can interact with its environment for extended periods of time without intervention. Just how long an 'extended period' justifies the term autonomous is dependent on the task at hand. In the MIT Autonomous Robot Design Contest, the robots function independently for 60 seconds and earn this title. If the domain is space exploration, this period may be much longer. An autonomous creature must be able to make decisions on its own, correct for unexpected circumstances, and be self-aware enough to know when a subsystem is not working perhaps even be able to fix it.

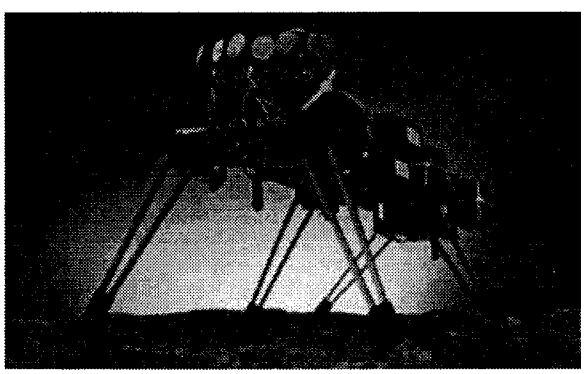

Figure 3.2 – Genghis, one of the first creatures

The definition goes on to claim that a creature must 'co-exist' in the world with humans. Existing in this manner has two implications for creatures. A creature must be embedded. An embedded creature does not deal with the world through simulation or abstraction; it deals with the world by interacting with it. A creature must also deal with the world in real time. A robot

driver that takes half an hour to figure out it is going to crash into oncoming traffic unless it turns to the left is an example of a system that fails this real time requirement.

The last part of the definition dictates that a creature must appear intelligent. Again, what constitutes 'intelligent' depends on the creature's purpose. By limiting the activities expected of the creature, satisfying this constraint becomes easier. Defining a baseline of intelligence requires constraints that apply regardless of the purpose of the creature. One aspect of intelligence we want from these creatures is that they respond appropriately to changes in the environment. The reaction need not be optimal, but it should be fitting and timely. A creature should be able to pursue multiple goals at once and when circumstances dictate, a creature should modify the goals it pursues, adapting to its surroundings. A creature should also be robust. Minor changes should not cause a catastrophic failure; rather one should expect that performance degrades gradually and gracefully as problems arise. Finally, a creature should accomplish something useful.

This definition of a creature is precisely what is wanted from the Cloud, or from any autonomous spacecraft. In this way we could populate outer space with many such creatures, let them roam without intervention, and simply collect the data they send back. We just need a design that accomplishes all these characteristics.

3.4 How creatures have worked

Two approaches to control systems for creatures have developed over the last 40 years, symbolic and behavioral. I present both these approaches and evaluate how well they meet the criteria presented above. Symbolic systems are the classic solution to problems in robotics, allowing fairly complex behavior in extremely simple environments. Behavioral systems are a newer solution to the problem, allowing robots to function in unconstrained environments but unable to accomplish complex tasks.

Figure 3.3 – Shakey

3.4.1 Symbolic Systems

The symbolic approach separates abstraction from reasoning from action. A robot that follows the symbolic paradigm abstracts sensor values into a representation of the world, reasons based on this representation, and takes action based on this reasoning. This approach is reminiscent of the Planning/Scheduling

approach taken by the NMRA. This leads to a clear separation between the sensor systems, the motor systems, and the reasoning engine.

The symbolic approach was the original solution in Artificial Intelligence to an embedded control system. Shakey, shown in Figure 3.3, was one of the first symbolic robots, built in the late sixties and early seventies [Nilsson, 1984]. Shakey lived in an extremely simplified world, where the only objects were painted polyhedral objects. In this environment Shakey was able to form and execute complex plans. She was considered to be a great success, demonstrating a system that successfully integrated many of the AI techniques of the day. As successful as Shakey was, robotics has progressed little since her days. To this day no robot has been created that can match Shakey's performance in a real world setting.

Shakey is typical of symbolic robots. These systems are capable of fairly complex behavior in very simple environments. When confronted with a complex environment the symbolic descriptions tend to break down and reasoning becomes unwieldy. Symbols are unable to capture the full complexity of the real world and unforeseen circumstances decrease the reliability of both the robot's plans and the robot's world representation. The overabundance of information slows the planning algorithms down greatly, just when quicker response times are needed. In extremely simple environments however, these systems behave beautifully; they can satisfy all of the constraints listed above.

3.4.2 Behavioral Systems

The alternative to the symbolic methodology is the behavior based approach. A behavioral system is one that is made up of many simple behaviors, complete in and of themselves, linked together in a way that is exploited by high level goals and/or a complex environment. This approach has its roots before Shakey was even imagined, but has only recently started to grow to fruition.

In 1949 Grey Walter began building artificial creatures he called tortoises (Figure 3.4) [Walter, 1953]. These creatures had only two vacuum tubes, two sensors (light and bump), and two actuators (wheels), but they seemed to exhibit "exploratory, speculative behavior that is characteristic of most animals". This was accomplished with only two basic behaviors: head towards the light when the sensor reports a value below a certain threshold, turn away from the light otherwise. With these two behaviors the tortoises

Figure 3.4 - Walter's tortoise

would wander the room, heading towards then away from the light, in a complex manner. More astonishing than this is that the tortoises exhibited emergent behavior. When the batteries on the creature were running low it would become less sensitive to the light, which was mounted above the tortoise's hutch where its recharging station sits. Since it is now less sensitive to the light, the tortoise would wander closer and closer to the hutch, eventually recharging itself. When the batteries reached a level sufficient to trigger the creature's *avoid* behavior, the tortoise would jerk awake and flee the light. Thus from these two simple behaviors emerge something that appears to be self-preservation.

This approach to robotics lay mostly dormant until around 1989 when modern behavior based systems took form. The basic idea remains the same, but is now pursued using modern engineering ideas and modern hardware. One of the first behavior based robots was Genghis, pictured in Figure 3.2 [Angle, 1989][Brooks, 1989]. Genghis had a total of 57 different basic behaviors, implemented as simple finite state machines (FSMs). These FSMs were linked together using Brook's subsumption architecture (also known as Brooksian architecture). They implemented robust walking and wandering behavior in a six-legged robot, something symbolic implementations have not been able to accomplish. Genghis could handle rough terrain, avoid obstacles, and walk on inclines.

Behavioral systems tend to achieve the real-time and embedded capabilities easier than symbolic systems. Behavioral systems are distributed by nature, having separate behaviors able to function independently of each other. Because of this they tend to be robust; when one behavior is unable to function there are others that will continue. It is also easy to implement multiple goals in behavioral robots; you simply have multiple behaviors active at once. The place where these systems suffer is in usefulness. Symbolic systems leverage off of the Artificial Intelligence research for the last 40 years, which has figured out how to plan complex strategies and accomplish seemingly high level goals. Thus within their simple environments they can accomplish some very useful functions. Behavioral systems have a much more limited repertoire. How to obtain complex high level behavior out of them is still an open question. Since they are highly parallel and distributed, classic serial programming accomplishes little. Programming parallel systems is a young field and not as well understood as serial systems. Mataric has built upon Brook's foundation by integrating world representation into behavioral systems [Mataric, 1992], incorporating a useful aspect of the symbolic approach, but this work has not equaled the control systems available to symbolic robots.

The behavioral approach has many qualities that should be in the Cloud. In fact, if complex behavior could be programmed in to such systems, it would result in exactly what is wanted from the receivers and autonomous spacecraft, reliable, robust, reactive, autonomous, and seemingly intelligent. The architecture presented in the rest of this thesis is an attempt at such a system.

3.5 How creatures should do work

We have seen the behavior we want from these systems, and have taken a look at how some systems have accomplished this behavior. Now we will establish some design constraints DAG Networks should fulfill. Inspiration for these constraints is drawn from influential ideas from Artificial Intelligence.

3.5.1 Nature

Nature provides us with the only example of an architecture that accomplishes everything we could want: the central nervous system of animals. What lessons can we learn this example? Many of these lessons couple well with ideas from AI, and are presented later. One lesson is important enough that it is presented on its own, the need for decentralization.

Our system should be decentralized. There is no CPU in the brain, computation is distributed across the cerebellum and the cerebral cortex and even the peripheral systems in a way we do not yet understand. Knock out a chunk of gray matter, and chances are a person could function almost perfectly. Certain systems in the brain are necessary for survival, but the destruction of any one system will not break all other systems except though secondary effects. This shows the robustness of decentralized designs. This is because behaviors are separated and only interface through non-critical connections.

Decentralized systems also tend to be more reactive. This is because there is no bottleneck in communication with a center; we do not need to wait for the system bus to be free. It is usually easier for a distributed system to evolve, as modifying one behavior can not imply having to modify all others. Distributed systems also tend to scale better; new behaviors can be added without modifying the whole system. All of these are huge wins for building intelligent creatures.

3.5.2 Society of Mind

The basic foundation for the DAG networks is taken from Minsky's Society of Mind [Minsky, 1988]. Two aspects of the Society of Mind should be integrated into the system. First is the idea of *agents* connected by functionality. An agent in Minsky's framework is defined to be "any part or process of the mind that by itself is simple enough to understand." Key to this concept is that since the process is understandable, we should be able to encode the process in an algorithm. These agents are connected based on functionality. If agent A needs the functionality of agent B to achieve its purpose, agent A is connected to agent B by the ability to activate it. By

thinking in terms of functionality, we can easily connect different behaviors together. From this idea we gain the constraint that behaviors should be agents, that is an understandable process able to be reproduced as an algorithm. Furthermore these agents should be connected to each other based on functionality. The nodes and connectivity of our system have been determined.

Second is the idea that thinking can be represented as patterns of activation. In the Society of Mind, intelligence emerges when high level goals activate some agents. These agents in turn activate other agents. This activation spreads and eventually the goal is accomplished. At any given point, the pattern of activation determines exactly what is going on in the mind. This idea tells us how goals should be represented in the system. They should be a set of behaviors to activate. Once active, these behaviors should accomplish the goal.

On closer inspection, this idea can be seen as an analog of biological systems. On a cellular level, each neuron can be seen as an agent connected to other agents, activation corresponding to the firing of that neuron. On a higher level, the brain is divided into areas specific to certain tasks, such as the visual cortex for handling the vision system and the inferior temporal cortex for object recognition. Each of these areas can be seen as an agent, but they also seem to be made up of agents themselves. The visual cortex, for example, is divided into various layers, each of which seems to correspond with an aspect of vision. The analogy of the Society of Mind fits the brain well.

3.5.3 Streams and Counter-Streams

All goals do not originate from some central goal repository. The environment can also inspire goals. For example, when your finger is immersed in flames a new high level goal is introduced in your system corresponding to removing your finger from the fire. This means that goals originate both from sensory input and high level cognition. The inspiration for how to merge these separate driving sources comes from Ullman's idea of streams and counter-streams [Ullman, 1996, 317-358].

Ullman presents this idea in the domain of vision. The goal of his system is to match images gathered by a camera with stored images of the same object or person. Bottom-up processing starts on the input image and works its way towards the stored image. Top-down processing goes in the other direction, starting with a stored image and attempting to match it with the input image. Each level of processing corresponds to different transformations the image can be subjected to.

This idea also gets its inspiration from nature. In the central nervous system, for just about every ascending pathway (towards the brain) there is a corresponding descending pathway (towards a peripheral system). As Ullman puts it, "If a visual area in the cortex sends ascending connections to another visual area higher up in the hierarchy of visual processing, then, as a

general rule, the second area sends reciprocal connections to the first." This fact seems to indicate that the central nervous system does indeed use both top-down and bottom-up processing. Whether this processing is of the form Ullman imagines or not is an open question, but the prospects of bi-directional computation are attractive.

This system can be generalized to provide us with the solution of how to control the patterns of activation. Top-down processing in this case corresponds to activation that begins with an internal goal of the creature. It represents conscious action. Bottom-up processing corresponds to activation that begins at the sensor level. It can represent both reflex actions and the understanding of sensory input. The transformation at each level is an information transformation. We already know that behaviors are linked based on functionality. The functionality of a behavior can be categorized based on the type of information it computes. As activation spreads from one node to another the type of information being computed changes. This provides us with the needed transformation.

For example, consider the simple graph in Figure 3.5. This network shows how we could implement Walter's tortoises in such a netwo k. As in animals, the sensor in this network is always active. It feeds an analog signal into an analog to digital converter. Comparitors test this value against a threshold and activate the correct behavior. This simple system will recreate the high level behavior the tortoises exhibited. The links in this diagram do indeed correspond to functionality, and each node is simple to implement.

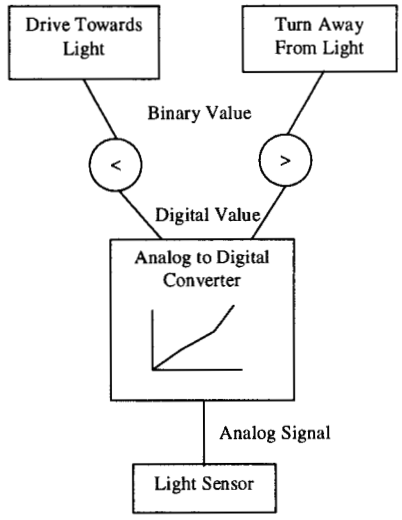

Figure 3.5 – Behavioral tortoise

As shown, this network behaves in a completely bottom-up fashion. This makes sense, as the tortoises were completely reactionary creatures. If we wanted to be able to have the tortoise be able to accomplish the goal "return to hutch" we can see how top-down behavior would work. This goal would simply activate the Drive Towards Light behavior and keep it active for as long as the goal remains. Since the hutch is right under the light, the tortoise returns home.

3.5.4 Routines

We have determined the general layout of the system, but the definitions we have so far could easily result in a symbolic type robot. To see this consider a system with three behaviors: Abstraction, Reasoning, and Execution (see Figure 3.6). According to the definitions so far, this network fits our design criteria. As robots such as Shakey have shown, these three behaviors are

simple enough for us to understand and program. They are linked according to functionality. They can easily function in both a top-down and bottom-up manner.

The problem with this representation is that the behaviors are too complicated. The result is a system that is less robust, less distributed, and less flexible. To avoid this we need some guidelines on how to design our behaviors. This guidance will also come from Ullman, from his ideas on visual routines [Ullman, 1996, 263-315].

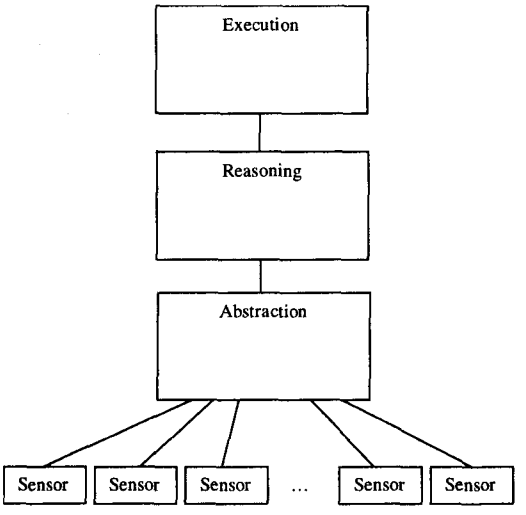

Figure 3.6 - Behavioral does symbolic

The idea behind visual routines is that there exists a set of elementary visual operations that can be combined in different ways produce routines capable of handling complex spacial tasks. These elementary operations should span the space of the tasks we want to accomplish. The application of these routines is determined both by the visual input and the specific task at hand. The output of these routines are incrementally put together to allow later processing to benefit from these results. This idea of a spanning set of elementary operations has proven to be extremely powerful and fits well the model we have developed so far.

Rao identifies two key issues that one must deal with when designing using routines [Rao, 1998]. One, what is a good set of elementary operations? Two, how do these primitives get strung together to perform a specific task? We are well on the way to answering the second question, and the answer to the first question is exactly the guideline we want for designing behaviors.

A real life example of routines in action comes from the eyes of frogs [Lettvin, 1959]. Lettvin identified four operations that a frog's eye seems specifically tuned to accomplish. He called operation 1 "sustained contrast detection". This operation activates when an object either lighter or darker than the background moves into sight and stops. Operation 2 is "net convexity detection". This operation seems to be a bug detector; it responds best to small object moving in a jerky manner across the frog's visual field. Operation 3 is a "moving edge detector" that activates most strongly when an object moves quickly through the frog's visual field. The fourth operation is "net dimming detection". This operation responds to a darkening of the frog's visual field. These four operations can be used by the frog to build routines to catch flies and avoid predators, the two most important high level goals to a frog.

The example of the frog shows that these basic operations should be fairly specific to the task at hand, making it difficult to make sweeping generalizations about what behaviors should be used in this control system. They should be simple, allowing complex behavior to emerge from

interconnection rather than being explicitly programmed. They should also be designed to provide a useful abstraction to other behaviors. When activated, the next higher level of activation should readily use the information they return.

3.5.6 One-Shot Learning

One of the most difficult problems intelligent control systems face is how to learn. Even if a system is capable of accomplishing complex behavior, how can it know how to accomplish it unless explicitly told? In the system we are developing this problem translates to how to learn goals, and how to learn what patterns of activation can lead to accomplishing that goal.

For this we turn to a powerful idea that has emerged in machine learning. Two buzzwords have become associated with this idea: One-shot learning and Near-miss learning. Such a system can learn from a single example, using further examples to refine the concept it is learning. One of the first systems to display such learning was Winston's Arches program [Winston, 1970]. This system was presented a description of an arrangement of block and then told whether that arrangement was considered an arch or not. By using differences between the stored representation of "arch" and the scene it was just told was an arch, the system would refine its description. After only a few example a correct conception of what an arch was emerged.

This style of learning has proven to be extremely powerful when it can be applied correctly. Two more modern examples of systems that employ one-shot learning to acquire grammar are Kirby's evolving population of learners [Kirby, 1998] and Yip and Sussman's phonetic knowledge acquisition engine [Yip, 1996]. One of the important ideas that emerged from Yip's treatment of the subject is that all one needs to accomplish one-shot learning is a high-level concept linked to a bit vector. In his program the high level concept is a word with the bit string representing the corresponding phoneme and meaning vectors.

We have a high level concept: the goal. The pattern of activation translates easily into a bit string. Thus our system should be able to employ one-shot learning in order to learn new goals. For example let us suppose we have a robot that knows how to drive straight and how to turn. We want to teach this robot the concept of "avoid". To do this we provide the robot with a positive example of what it means to avoid; we tell it to turn away when sensor1 reads too high. At this point the robot sees what activation led to "avoiding" and remembers that it should turn away when sensor1 reads too high. What we want is a more general avoid behavior though, not one tied to sensor1. Thus we provide the system with another positive example; this time when the robot turns away when sensor2 read too high. One-shot learning would take a look at the difference between these two examples, and see that it does not matter which sensor read high, simply that some sensor reads high. Thus avoidance becomes linked to the pattern of activation that corresponds to turning away when some single sensor value crosses a threshold. Further

learning could teach the system how to "avoid X", linking the concept with the sensor and threshold needed to avoid something specific.

3.6 Summary

This chapter described a system for building control systems for intelligent creatures. An *intelligent creature* is defined as an autonomous agent that co-exists in the world with humans and is seen as an intelligent being. There have been two frameworks for such systems in the past. Symbolic implementations rely on reducing the world into symbols, computing based on those symbols and internal state, then acting on these computations. Behavioral implementations have developed to address some of the shortcomings of symbolic systems. A behavioral system connects together simple behaviors and relies on the complexity of the environment to provoke intelligent action.

Ideas from successful systems motivate what the control system should be like. They lead to a system based on patterns of activation of simple, yet expressive behaviors. This system uses both bottom-up sensor inputs and top-down goals to determine how behaviors are to be activated. One-shot learning can be used to match these high level goals with matching patterns of activation.

Chapter 4. The Control System Design

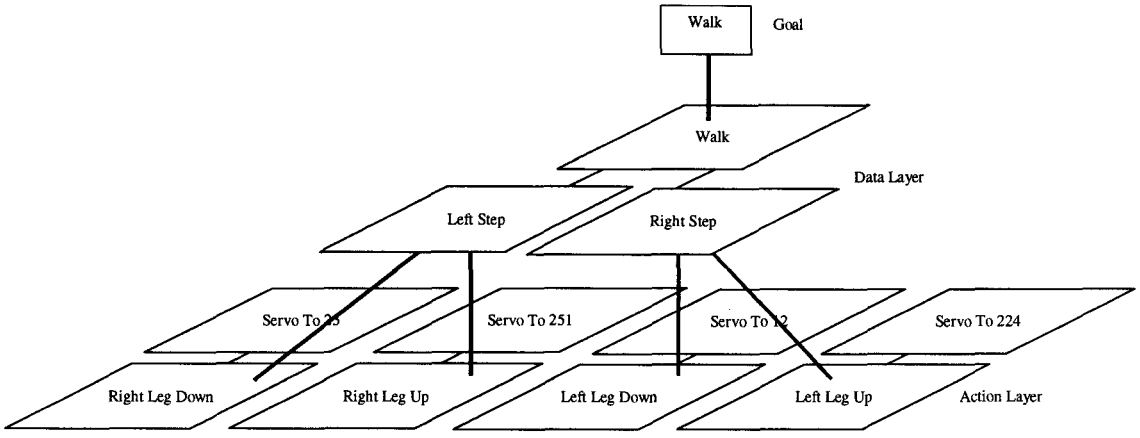

Figure 4.1 – An example DAG network

4.1 An outline of the design

The previous chapter provided many guidelines for how to design this control system. In this chapter these pieces will be put together into an architecture capable of meeting the requirements set forth. The design consists of two layers. The top layer is the data layer; its purpose is the manipulation of data. There are two types of elements in this layer; sensors and data behaviors. The lower layer is the action layer, where all the interfaces to the actuators lay. All the elements in this layer are action behaviors that control the "motor" functions of the receiver and the actuator interfaces. Data behaviors can activate action behaviors, which is the only way the two layers can interact.

Hovering nebulously over these layers is the goal engine. This is where the mappings from goals to patterns of activation are kept. Currently active goals activate various behaviors, igniting the robot's processes. Data behaviors have the ability to both activate and suppress goals. Figure 4.1 shows an example network that is part of the network for a six-legged robot that allows the robot to walk.

4.2 The data layer

4.2.1 Data behaviors

Data behaviors are small functional units. They take in arguments and produce an output, just as in normal functions. They execute serially, just as most programmers are used to dealing with. They can have state while activated. The complexity of a data behavior is not limited by the system, though a good behavior should be simple, small, have limited dependencies, and have a small constant running time. These features, which match how programmers program, should make it easy for developers to design data behaviors.

There are differences between behaviors and typical functions. In a behavior you are rarely passed arguments directly, you obtain them by either querying a sensor or activating another data behavior. This is because you do not know if you were activated in a top-down fashion or a bottom-up fashion. If bottom-up, then the data you need for your computation should be readily available, but if somebody higher up the hierarchy activated you, the data you require might not be available and further computation will be needed. By enforcing this rule, all connections in the layer are made two-way.

A request for information activates those behaviors able to provide it. It can be made either asynchronously or synchronously. In the asynchronous case the behavior does not wait for the data, but continues executing. A signal is sent to the requesting behavior when the data becomes available. In the synchronous case, the behavior waits for the data before continuing execution.

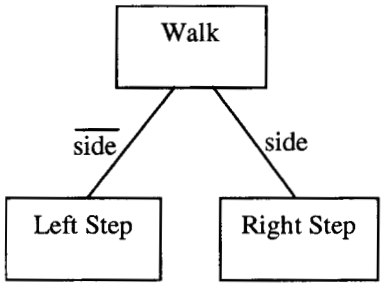

Figure 4.2 – Walking

For example, let us consider the walk behavior from Figure 4.1 and Figure 4.2. This behavior may implement a very simple walking routine, take a step with the left side, then one on the right, repeat. It is dependant upon the two stepping routines, but does not care how "stepping" is carried out. Some response from the step behaviors may be expected, perhaps a failure condition. To keep this example simple, the assumption is that the behaviors do not return any data. In the Figure 4.1, the walk behavior is activated directly from a goal, but could just as easily been activated by another behavior such as *pace*. Pseudocode for this behavior is:

```
boolean side;

side := not(side);
if(side) get(right step, synchronous) else
         get(left step, synchronous);
```

As long as the "walk" goal is present, the walk behavior will be active. While it is active it has state as represented by the *side* bit. This bit represents whose turn it is to take a step, the left side or the right. The behavior simply takes the correct action. As long as the behavior remains active, it will loop, flipping side and taking alternating steps.

4.2.2 Sensors and Sensor behaviors

Sensors are a special case in the data layer. A sensor is connected to a data behavior (which may have multiple sensors attached to it), which is called a sensor behavior. The special aspect of sensor beh viors is that they may always be active. This is a necessary condition for bottom-up processing.

Consider the example of a thermometer as a sensor. A robot may want a behavior that removes this sensor from its location if the ambient temperature crosses some threshold. In this case we would want the sensor behavior associated with the thermometer to always be active and always checking the threshold. If the temperature exceeds the threshold it can set a reaction in motion which will remove the sensor.

4.2.3 Connections between data behaviors

Connections occur between behaviors when there is a data dependence between the two layers. When one behavior needs the information that the other behavior computes, or needs the action that the behavior results in, it is linked to it by the power of activation or suppression. Usually these links are conditional, they are not used each time a behavior is activated. They depend on the state of the system at the time and the internal state of the behavior. Sensors cannot be dependent upon behaviors for activation. They are always active by default. The sensor behaviors connected to the sensors may be deactivated, waiting for the system to want an interpretation of the sensor input.

There are two types of activation for data behaviors. Some behaviors, such as the step behaviors above, should be activated, execute once, and then deactivate. The tasks they perform are done and if we want the system to take multiple steps, the step behavior should be activated multiple times.

Other behaviors should stay active until some other agency deactivates them. The walk action is an example of this. A walking behavior should not just take two steps, it should continue taking steps for as long as walking is desired. These are continuous behaviors, which do not deactivate after one execution but loop until deactivated.

4.2.4 Connections to the action layer

The links to the action layer are similar to links between data behaviors. They can be synchronous or asynchronous. They can be one-shot or continuous. The difference between these links is that links to the action layer may carry arguments. This is because action behaviors cannot activate data behaviors, and thus have no way of requesting extra information. By allowing arguments to be passed to action behaviors we allow the data behaviors to provide some of the abstraction of the actuators.

There is biological evidence that there are complex transformations that change motor plans into motor commands [Bizzi, 1991]. Argument passing allows part of this transformation to take place in that data layer where reasoning and planning is meant to occur, and part to take place in the action layer where execution is meant to occur. Thus the full power of the system is available for controlling actuators.

4.2.5 Data behaviors and goals

The data layer is the only part of the system that interfaces with the goal engine. This is a two-way interaction: the engine sets up patterns of activation in the data layer, and data behaviors can activate and deactivate goals. A goal is associated with certain behaviors. It may be a simple relationship, such as the walk goal simply activating the walk behavior or a more complex relationship, such as a "pat head and rub belly" goal.

The connection goes the other way as well; data behaviors can instantiate and suppress goals. This is a powerful mechanism that allows the system to adapt to its environment and react in complex ways. Consider the example of Walter's tortoises. One way of getting them to return to their hutch would be to have a battery low sensor and have its associated sensor behavior activate a return to hutch behavior. When the battery low indicator turns off, the return to hutch behavior is deactivated and everything works out well. Another way of accomplishing the same thing would be to have the sensor behavior instantiate the get power goal. This goal in turn would activate the return to hutch behavior. When the sensor no longer reports a low battery this goal can be removed.

The second way is a much more robust way of achieving the desired result. First, it provides a layer of abstraction between the low battery signal and the return to hutch behavior. By having a get power goal, we allow that the world may change or that there may be multiple ways of getting power. If the power supply were no longer located in the hutch, the second method would require less work to rewire the system. Second, it allows for other behaviors to take precedence over the return to hutch behavior. If the robot were in the middle of executing a complex task, we would not want it to stop if its power situation were not critical. The direct connection should be used for reflex actions, ones the robot should have no control over.

4.2.6 More on bottom-up activation

Top down activation proceeds in a straightforward manner. A behavior needs information it does not have so it activates behaviors that are able to produce the information. How bottom-up execution is to progress is a little more complicated. If activation were to spread in a way analogous to top-down activation, by activating all possible behaviors that can use the information produced, we would soon be swamped by activity not relevant to anything at hand. Consider how many tasks vision is used for. If each of these tasks were always active, we would be overloaded by doing too much at once.

A general way to solve this problem is to associate with each behavior a list of behaviors that it should activate upon completion. I will refer to this list as the propagation list. Any behavior or goal that activates a specific behavior may modify that behavior's propagation list. Behaviors also have an intrinsic propagation list that cannot be modified. This intrinsic list can be used to define a default bottom-up activation scheme for the behavior. The variable list allows bottom-up processing to occur as a result of other activation.

The variable list has various other. It provides a natural way of controlling what is a one-time activation and what is a continuous activation. If you want to activate a behavior in a continuous fashion, you simply add that behavior to its own propagation list. When the behavior terminates it will start itself back up. It also provides a natural way of implementing callbacks and looping behavior. Later I will show how this can be used to help avoid problems inherent in parallel systems.

4.2.7 Parallel execution

This brings us to the idea that multiple goals can be active at one, multiple data behaviors will be active at once, and multiple action behaviors will be active at once. This results in various complications to the basic scheme. How do data behaviors cope with this complexity?

One complication that arises is that multiple behaviors may influence a single behavior, call this single behavior B. Some set of behaviors may want B active and another set may want to keep B from becoming active. Should B be activated or not? Another similar problem is what happens when multiple behaviors want to effect an actuator in different ways. If one behavior wants the leg down and another wants it up, what should the leg do?

The answer to this problem is to assign a priority to tasks. When a goal is activated it is assigned a priority, either by the goal engine or the behavior that activated it. This priority is given to behaviors it activates. If multiple nodes want to activate a behavior, that behavior takes on the highest priority amidst its activators. If there are nodes that want to deactivate an active

behavior then some kind of resolution scheme must be implemented. What this scheme is can be left up the implementation, but it must exist.

Another type of problem that can arise in parallel execution is deadlock. Deadlock can happen when one behavior is waiting on a behavior that is waiting on the first. In order for this to be a problem, two behaviors in a loop must be waiting on each other synchronously. Consider the partial network shown in Figure 4.3. Here, if A is activated it will activate B then wait for B to finish its execution. B will then attempt to activate A, and then wait for A to finish, which can now never happen.

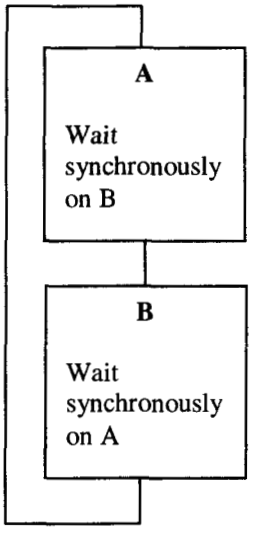

Figure 4.3 - Deadlock

The situation is even more complicated in the face of bottom-up activation. If A was activated in a top-down fashion, and B was activated in a bottom-up fashion, deadlock may happen even when there is no loop in the picture.

There is no general runtime solution to such deadlock. Rather, the designer of the network must use some kind of locking scheme for behaviors that make synchronous calls, ensuring that such situations never arise. It is interesting that the brain, a massively parallel computer, does not seem to ever encounter deadlock. It seems that there is no such thing as a lock or a synchronous call in nature, and thus no danger of deadlock. In this type of network, it seems we should take a cue from nature and avoid synchronous calls whenever possible. Unfortunately, this likes in the domain of the design of the specific network, not in the design of the architecture.

4.2.8 Deactivation

It is clear how a behavior should be activated, simply start executing. Deactivation is a little more complicated. Not all the activities associated with a behavior are contained within that behavior. For example, in the walk network described above even if the walk behavior is left active, one of the step behaviors will continue executing. To avoid this situation deactivation should be propagated down all the behaviors the deactivated behavior can activate. This propagation is done at a low priority. Thus, if other behaviors are currently executing the behaviors this propagation reaches, the low priority will keep them from deactivating. If the only activation for a behavior comes from the deactivated behavior, then the active behavior should be deactivated no matter what.

When a behavior is deactivated, if the priority of the deactivation is enough to warrant deactivation, the execution of the behavior is halted right where it is and the deactivation is

propagated on to other behaviors that may have been activated. This deactivation is also passed down to the action layer, and may have to be passed into the goal engine.

4.3 The action layer

The action layer is the part of the system that interfaces with the actuators. Elements of this layer are either action behaviors or actuator interfaces. The purpose of the action behaviors is to abstract away the complexity of dealing directly with actuators.

4.3.1 Actuator interfaces

Just what an actuator interface is depends greatly upon the sensor. To move an arm about, a certain pulse width modulated signal might have to be sent to a servo. To turn a wheel, perhaps a certain current must be supplied to a motor. Perhaps a value stored in a register controls the hardware that accomplishes these low-level actions. In this case the actuator interface would be storing a value in that register. Whatever the interface, it provides a software level method of controlling the actuator.

4.3.2 Action behaviors

The purpose of action behaviors is to transform the actuator interfaces into higher level interfaces that the data layer will want to deal with. Action behaviors act very similarly to data behaviors, but in a simpler fashion. They do not have to communicate with any other layer or the goal engine. They do not have bottom-up processing, for as far as we currently know, muscles do not initiate any processing.

Though this layer is simpler than the data layer, the task action behaviors have to accomplish is not simple. Developing a high-level language for controlling actuators in an unconstrained setting is still an active area of research. The systems that currently exist are extremely complex. One of the most successful has been HANDEY [Lozano-Perez, 1987]. HANDEY is an example of a symbolic system. It reduces the world to a *configuration space* and is able to produce complicated plans such as telling a robotic arm how to grasp an object, pick it up, put it down in a new position so that it can re-grasp it in a way so that it could be moved to a terminal configuration.

Almost all such systems require at least fairly complex mathematics in order to translate an end position for an arm into a set of actuator settings. This is known as the inverse-kinematics

problem. An interesting biology inspired approach that avoids the inverse-kinematics problem is presented by Bizzi [Bizzi, 1991]. This approach suggests that there is a coarse map of limb positions in the pre-motor areas of the spinal cord. Vector combinations of motor outputs from different areas of this spinal map produce complex motor behaviors. This approach was one of the inspirations behind the idea of an action layer. The pre-motor area of the spinal cord corresponds to action behaviors, which can easily compute such vector combinations. Once an action behavior is activated with an argument that describes a position for the arm to be in, a system similar to one Bizzi describes can position the arm. The hope is that since nature found such an elegant solution to the inverse-kinematics problem in the case of motor systems, other systems will have similar solutions. The action layer is a fundament that encourages such formalisms. Other than in the ways mentioned above, the action layer works just as the data layer.

4.4 The goal engine

The goal engine has three main purposes: keeping a list of active goals, mapping goals to sets of behaviors, and learning new goals. It is this list that is the interface between behaviors and goals. The list of active goals is the activities the system is currently engaged in. A behavior can add to the list and delete from the list. When a goal is added to the list the process of activating all the corresponding behaviors is started. When a goal is deleted from the list the process of deactivating all the corresponding behaviors is started. There are few complications behind either this or the mapping. The learning system is more complicated.

As the previous chapter said, the goal engine was designed to allow for one-shot learning of new behaviors. A complete description of the mechanics of one-shot learning is tangential to the purpose of this paper, but Winston's description of his ARCH program covers the subject beautifully [Winston, 1970]. This learning will allow the goal engine to learn new goals, which are essentially a list of activations associated with the activity being learned.

Currently the learning aspects of the goal engine remain pure theory. One-shot learning should be implementable, given the setup, but it is not a critical part of the system. It is a feature that will be added at a later date.

4.5 Example Networks

Now that the complete system has been described, I will present a few example networks. Three examples will be presented, first is a more complete treatment of the walking example given above. This shows how basic functionality can be implemented under the system. Next I will show how easily this architecture can replicate Brook's subsumption architecture by building

a network similar to the walking routines from Genghis. This shows both the ability of this architecture to simulate subsumption architecture and an example of a network that exhibits complex emergent behavior. The last is a simple example of how DAG Nets can be used to compute GPS solutions.

4.5.1 A walking network

Teaching a six-legged robot to walk is not a difficult task. Teaching a six-legged robot to walk robustly in a complex environment is something that few systems have satisfactorily accomplished. In this section I present a network that allows a six-legged robot to walk. This is a simple walking routine, but is modular enough that it can be made robust without great effort.

Legs on this robot have two servos, the A servo which controls the forward-back motion of the leg and the B servo which controls the up-down motion of the leg. Each servo has a

```
boolean side;

side := not(side);
            if(side) {
      add_propagation(right step, self);
      get(right step, asynchronous);
} else {
      add_propagation(left step, self);
      get(left step, asynchronous);
}
```

Figure 4.4 - Asynchronous walk behavior

```
get(rleg1_up, asynchronous);
get(rleg1_forward, asynchronous);
get(rleg3_up, asynchronous);
get(rleg3_forward, asynchronous);
get(lleg2_up, asynchronous);
get(lleg2_forward, synchronous);

get(rleg1_down, asynchronous);
get(rleg3_down, asynchronous);
get(lleg2_down, synchronous);

get(rleg1_back, asynchronous);
get(rleg3_back, asynchronous);
get(lleg2_back, synchronous);
```

Figure 4.5a - Right step behavior

```
get(lleg1_up, asynchronous);
get(lleg1_forward, asynchronous);
get(lleg3_up, asynchronous);
get(lleg3_forward, asynchronous);
get(rleg2_up, asynchronous);
get(rleg2_forward, synchronous);

get(lleg1_down, asynchronous);
get(lleg3_down, asynchronous);
get(rleg2_down, synchronous);

get(lleg1_back, asynchronous);
get(lleg3_back, asynchronous);
get(rleg2)back, synchronous);
```

Figure 4.5b - Left step behavior

corresponding register whose value moves the servo to an absolute position. The action layer in this case is very simple. Each leg has two action behaviors, set_A and set_B that simply replace the contents of the corresponding register with the argument. Each servo has an associated sensor, get_A and get_B that returns the current position of the servo.

There are a few constants built into the system: leg_up, leg_down, leg_forward, and leg_back. Each constant is the value to place into the correct register to move the leg into the corresponding position. The walk behavior is very similar to the one presented above, but takes into account the warnings about deadlock. Figure 4.4 shows this behavior. The right step and left step behaviors are simple procedures that explicitly move the legs in the correct manner for walking. Figures 4.5 a & b show the source for these behaviors. A synchronous action behavior call is made when we need all the above calls to finish before continuing. There is an implicit assumption here that the synchronous calls are not faster than the calls above them.

All that is left is to implement the various leg positioning calls such as rleg1_up and rleg1_forward. These are simple behaviors that call the correct action behavior with the appropriate constant, such as rleg1_set_B with leg_up for rleg1_up. Once that call has been made the behaviors wait until the corresponding get_A or get_B sensor behavior returns a value equal to the argument, thus making sure the servo has reached the correct position.

Part of the network for this system is shown in Figure 4.6. This figure only shows the network corresponding to one of the step behaviors, but the other half of the network is symmetric. The walking implemented by this network is not very robust. If one of the legs were blocked from achieving their position or if there were a hole beneath one of the legs, the network would never know. Because each of the leg positioning behaviors is its own module, they could be made more robust and more complicated easily. The next section shows an example of a more robust walking network.

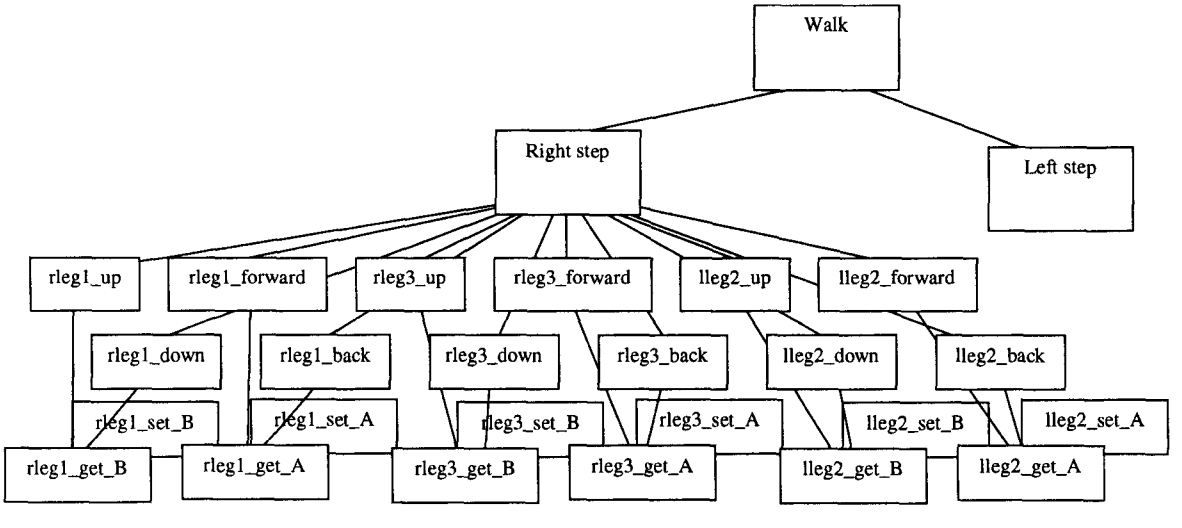

Figure 4.6 - A simple walking network

4.5.2 Genghis' walk

The example in this section shows two things. First of all, it mimics the walking behavior of Genghis, showing that DA Networks share the power of subsumption architecture. Second, it is an example of a fairly robust walking design, taking advantage of emergent behavior. Each behavior in this section exactly mimics one behavior from the network to control Genghis [Brooks, 1989].

The basic behavior in this system is standing. This is accomplished by having a stand data behavior as depicted in Figure 4.7. This behavior simply sets each leg to a default that is a standing position.

The next step is to implement a simple walk. This requires 5 different types of behaviors. These 5 behaviors work together to produce various simple gaits.

1) Leg down sensor behaviors. These sensor behaviors sit on each of the get_B sensors. They are triggered when the get_B sensor indicates that the leg is not in the down position. When this is sensed, the leg is to the down position. This forms six independent negative feedback loops, one for each leg's B motor. These behaviors correspond to the *leg down* machines in Genghis. Figure 4.8 shows an example leg down sensor behavior.

2) Alpha balance behaviors. This is the feedback loop for the A motors and it involves all 6 legs simultaneously. This behavior monitors the position of the A motor for each leg, treating

```
rleg1_set_A(A_init);
rleg1_set_B(B_init);
rleg2_set_A(A_init);
rleg2_set_B(B_init);
          ...
lleg3_set_A(A_init);
lleg3_set_B(B_init);
```

Figure 4.7 - Stand data behavior

```
if(rleg1_get_B != leg_down)
      rleg1_set_B(leg_down);
```

Figure 4.8 – leg down sensor behavior

```
sum := rleg1_get_A + rleg2_get_A +
       rleg3_get_A + lleg1+get_A +
       lleg2_get_A + lleg3_get_A -
       (6 * leg_middle);
correction := sum / 6;
rleg1_set_A(rleg1_get_A + correction);
rleg2_set_A(rleg2_get_A + correction);
rleg3_set_A(rleg3_get_A + correction);
lleg1_set_A(lleg1_get_A + correction);
lleg2_set_A(lleg2_get_A + correction);
lleg3_set_A(lleg3_get_A + correction);
```

Figure 4.9 – Alpha balance behavior

straight out as zero, forward as positive and backwards as negative. It sums these six values and tries to set each so that the sum is zero. It does this by sending an identical message to all six set_A action behaviors, which moves each leg an amount in the needed direction to move this sum to zero. Thus if one leg happens to move forward for some reason, all legs will receive a series of messages to move backward slightly. This behavior stays active. There are a few ways to accomplish this. This behavior could be in the propagation list of some of the sensor behaviors. It could be started up in continuous mode by some other behavior or goal. Or it could be activated once and then add itself to its own propagation list. This example simply makes it a sensor behavior. Figure 4.9 shows the psuedocode for this behavior.

3) Alpha advance behavior. There are one of these behaviors for each leg. It monitors the position of the B motors and is triggered when its leg is raised. When triggered it forces the leg forward. It does this at a higher priority than the alpha balance behavior, thus while this behavior is active the corresponding leg moves forward and the alpha balance behavior moves all other legs back. When it is finished the leg down behaviors will bring the leg back down to its normal standing position. The source for this behavior is in Figure 4.10.

4) Up leg trigger. This behavior lifts a leg when it is activated. It does this at a priority level high enough to overcome the leg down behaviors. There are one of these behaviors for each leg. With this behavior the robot can very nearly walk. If an up leg trigger behavior is activated, it lifts its leg, which triggers the alpha advance behavior. This swings the leg forward and all other legs swing back, moving the robot forward. The leg down machine will then put the leg down. The psuedocode for this behavior is shown in Figure 4.11.

5) Pattern generator. This is the final piece of the machinery for walking. Its purpose is to sequence walking by activating the up leg trigger behaviors in an appropriate pattern. In Genghis, two patterns were used. The first resulted in the alternating tripod gait, the same gait the first walking example used. Activating lift triggers to appropriate leg triples simultaneously accomplished this. The second produces a standard back to front ripple gait by activating legs in series. Changing this behavior easily changes the robot's gait. This behavior should be active for as long as we want the robot to walk. The code that produces the tripod gait is reproduced in Figure 4.12

These behaviors are enough to result in a simple walking activity similar to that accomplished by the previous example. Genghis took this behavior a little further with the help of a few more sensors and a few more behaviors. There are 21 more sensors in Genghis: a force sensor for each motor on each leg, two "whiskers", an inclinometer, and six infrared sensors. Genghis also has 6 more behavior types.

Force balancing helps compensate for rough terrain. The force that motor B is exerting is monitored, if it crosses a threshold the leg is held where it is. The rational behind this is that if a leg is resting on an obstacle and pushing down, it will be trying to lift the rest of the robot, resulting in high torque on the motor. In this situation the leg should be left where it is, as it is already providing enough support for walking. There are 6 of these behaviors, one for each leg.

Leg lifting is another mechanism for handling rough terrain. These behaviors monitor the force on the A motors. In general you want to lift the leg as little as possible to speed up walking. This idea fails when there is an obstacle in front of the leg. If there is something keeping the leg from swinging forward, these behaviors sense it and attempt to lift the leg higher. There is one of these behaviors for each leg.

Feelers are the behaviors that use the information the whiskers provide. If the whiskers detect an object in front of the robot, these behaviors make it so that the legs on the appropriate side are lifted higher on the next step cycle. There are two of these behaviors, one for the right whisker and one for the left.

Pitch stabilization is a behavior used to compensate for force balancing during times it is the wrong thing to do. When the whole robot is pitched forward, there will be more force on the forelegs. When it is pitched backwards, the rear legs will see a heavier load. This would cause force balancing to keep these legs from being lowered all the way, causing the robot to sag and increase the pitch even more. These behaviors, one for forward pitch and one for backwards, suppress force balancing under these conditions.

Prowling is a simple behavior that keeps the walking behavior inhibited until the IR detectors sense motion. It is a unique behavior that takes its input from the IR sensors and activates when it senses a change.

```
if(rleg1_get_B != leg_down)
      rleg1_set_A(leg_forward);
```

Figure 4.10 - Alpha advance behavior

```
      rleg1_set_B(leg_up);
```

Figure 4.11 – Up leg trigger

```
get(rleg1_up_trigger, asynchronous);
get(rleg3_up_trigger, asynchronous);
get(lleg2_up_trigger, asynchronous);

sleep(1.2); /* Wait 1.2 seconds */

get(lleg1_up_trigger, asynchronous);
get(lleg3_up_trigger, asynchronous);
get(rleg2_up_trigger, asynchronous);

sleep(1.2); /* Wait 1.2 seconds */
```

Figure 4.12 – Pattern generator for an alternating tripod gait

The final behavior is steered prowling. The IR detectors tell this behavior what side of the robot the movement is coming from. It then shortens the length of the backswing of legs on that side of the robots. This has the effect of turning the robot towards the source of the motion. Thus the robot can follow slowly moving robots.

These behaviors complete the control system of Genghis. Each of them translates directly into DAG behaviors like those described in the basic walking example. This shows that DAG networks have at least as much power as subsumption architecture. They can describe systems just as robust and distributed. The links between elements allow complicated emergent behavior to result from the basic behaviors, just as in subsumption architecture.

Note that Genghis is a purely reactive system, as is each of its components. Because of this there are no goals in this system, nor do any of the behaviors return any values. The action layer components of the system were extremely simple. This shows that the Genghis example did not utilize the full power of the system, yet it was able to exhibit fairly complex behavior.

4.5.3 A simplified GPS solution

This section presents a simplified network for computing a GPS solution. The system in this example ignores many of the complexities that would actually arise in computing such a solution. The needed calculations are simplified, and the receiver in the example has only four channels, just enough to track the four satellites needed for a solution. These channels are the sensors in the example, whereas in an actual receiver lower level sensor inputs would probably be available. Also, the action layer is extremely simplified. The UART is simply something that we can send positions to that will do the right thing, and our access to a *satellite tracker* (such as the ASICs on the Rogue receivers) simply consists of telling it how to search for new satellites.

The key behavior in this network is the Get Lock sensor behavior. This behavior is

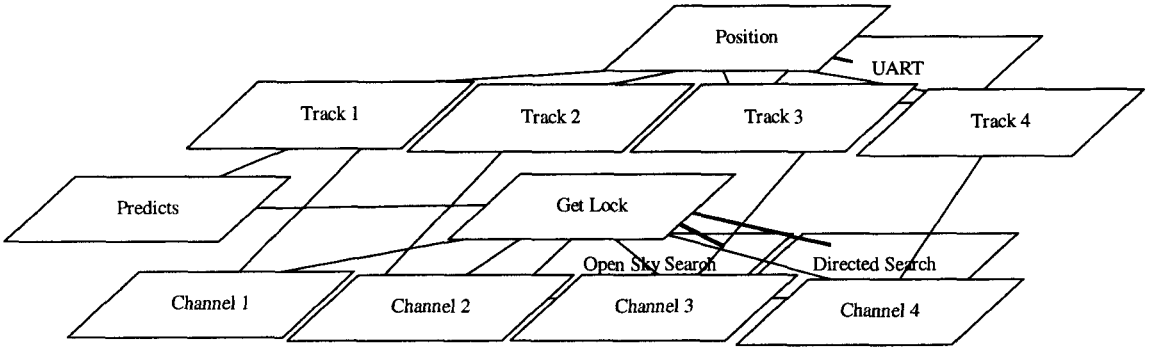

Figure 4.13 - GPS DAG Network

activated whenever any of the channels report not having a satellite to track. When satellites are needed this behavior takes a look at whether Predicts are available. Predicts are approximate positions of the satellites gotten from an almanac of a satellite that was recently tracked. If predicts are available the Get Lock behavior will active the Directed Search action node. This will tell the satellite acquisition system what satellites to look for, making the acquisition process fairly quick. If predicts are not available then an Open Sky Search (OSS) is initiated. An OSS searches the whole sky for whatever satellites can be found, and thus can take a fairly long time.

Once satellites have been acquired, the Track behaviors will take the data stream from the channels and compute the distance to that satellite. Once this has been done for all four satellites, the Position behavior takes this information and calculates the GPS receivers position. It then sends this position to the UART which send the information out the data port to whoever is listening.

This is an extremely simplified implementation of the GPS process, but it shows a framework for how the solution can be implemented using DAG Nets. Furthermore this formulation is easily extendable by adding more behaviors. For example, monitors can be added to the channels to detect when they are not working, allowing the system to notify ground control or perhaps even take corrective action.

Chapter 5. Contributions

This section will evaluate the contributions of the communications library and the DAG architecture. The communications library will be evaluated on its usefulness and functionality. DAG Nets will be evaluated along various dimensions. First, it will be compared to symbolic systems in general and to subsumption architecture. Then we will evaluate how well DAG Networks meet the criteria set forth in chapter 3. Once this is done, ideas for possible future work will be given and discussed.

5.1 The communications library

The communications library is already in use on the next generation Rogue software. The functionality of the system has met the needs of the programmers, except for applications that need extremely high-speed communications. It has proven to be easy to use and because the interface closely resembles that of the previous communications, little work was needed to incorporate it into existing software.

5.2 DAG Networks compared to symbolic systems and subsumption

The first thing to note about DAG Networks is that it is a universal framework. The fact that behaviors in the network are not restricted means that anything can be encoded as a single behavior that is always active. Action nodes can be the simple function calls that are the low level actuator interface. This way of using DAG Networks does not exploit any of the power that lies under the surface, but in the case that an activity cannot be broken down into small, parallel executing pieces with simple interactions, it is a backup.

Figure 3.6 showed one of the simplest DAG Networks that captures the symbolic solution. The nice thing about DAG Networks is that it is a framework that makes it easy to take advantage of any breakdown of the three basic symbolic systems. For example, you can have the sensor behaviors constantly update the internal world model in the robot. The planning system can simply use the most recently updated version. In this way the perception step becomes more distributed and can exploit parallelism. Systems like DYNORAII [Shekhar, 1993] show that even the planning step of symbolic systems can be optimized for real-time systems. As for the execution phase, there are many physical actions that can be taken at one. DAG Networks make

exploiting this easy, so that even simple execution phase implementations may see a performance boost. These are the ways DAG Networks have an advantage over symbolic systems.

Symbolic systems are usually custom made to the task at hand. Because of this they can be optimized for hardware, software, and environment. If a DAG Network is used for symbolic systems, it must pay the cost for having the DAG system work in the background. The kind of parallel processing and activation that DAG Networks offer symbolic systems also may not be quite right for the task. In this case a symbolic system would have to settle for a less than optimal solution to fit into the DAG framework. In this way DAG Networks are weaker than symbolic systems.

Subsumption architecture shares many features with DAG Networks. Both are distributed systems. In subsumption architecture all behaviors are encoded as FSMs whereas DAG Networks use rbitrary programming languages. This makes subsumption architecture more appropriate for applications where hardware versions of these FSMs can be fabricated. Custom fabbed circuits will be faster than compiled code. The compiled code has the advantage of expresability over FSMs. In subsumption architecture behaviors are active unless suppressed by another behavior, in DAG Networks they have to be explicitly activated (unless they are a sensor behavior). There are no explicit goals in subsumption architecture; goals emerge from the various behaviors. Subsumption also have very little state, since it wants to remain close to the hardware, the only state in pure subsumption architecture is contained in registers. In DAG Networks, arbitrary structures can be stored.

Subsumption has the advantage of speed and closeness to hardware. It is the assembly language equivalent of DAG Networks. DAG Networks on the other hand are much more expressive. As was shown in the second example, DAG Networks have the power to represent subsumption. DAG Networks are also further removed from hardware than subsumption. Subsumption systems can be built out of simple hardware components while DAG Networks require processors, memory, and all the associated hardware. One advantage to DAG Networks are that processors and memory are getting cheaper and cheaper. The system is arranged such that it should be easy to build a multiple processor version. Thus DAG Networks may eventually share the hardware advantage that subsumption now enjoys. The fact that any processor can implement any behavior in DAG Networks is another advantage it has over subsumption systems. Behaviors in DAG Networks are simple code; they can be changed and interchanged without hardware changes. Since subsumption deals with much hardware on a much lower level, such changes are harder (unless an interpreter is being used at which point subsumption has given up its speed and simplicity advantages).

5.3 Do DAG Networks meet the criteria?

The main criteria set forth for DAG Networks was that the task of computing a GPS solution could easily be fitted into the framework presented. Again, at the very least, the universality of the architecture means that implementing a GPS solution will be at least as easy under DAG Nets as under a more classical foundation. Moreover, the task of computing a GPS solution can be broken down into smaller parts, some of which can easily be done in parallel. An example of tasks that can be computed in parallel is the first stages of computation on the data streams from each satellite. These data streams do not depend on each other until later in the computation, so DAG Nets should easily be able to exploit this parallelism.

The system should also encourage reactivity, robustness, and a distributed implementation. The partitioning of the task into relatively independent behaviors satisfies all of these criteria. The reactivity of the system stems from how sensor behaviors can propagate sensor input, so that action can be taken immediately upon sensing some condition. Robustness and distributed-ness come from the fact that behaviors are by their nature distributed. As long as the task is broken down into small enough parts, the result will be distributed. If these parts are not dependent upon each other, then the system will be robust, as if any one part breaks down all behaviors not dependant upon the broken behavior can keep on functioning normally.

5.3 Future Work

The communications library is functional as it stands. More functionality can be added as needed, but there is no future work currently planned. There is much work to be done on the Cloud. Currently the implementation of DAG Nets is undergoing revision and debugging. The design is as presented in this thesis, but the interface is still in flux and being debugged. Getting a stable implementation of the base system is the first priority.

Once a stable release is ready, it will be used on the Rogue receivers. Further revision and debugging is expected when developers begin using the system. The result of this phase will be the system that is ready for use in the receivers. This first system will be a fairly simple implementation, without code to monitor the state of the system or to take much initiative in what actions it takes. It will follow the classic paradigm of having a completely sequential program for the receiver to follow. Later systems will add monitor code and more reactive programming, allowing the receiver to behave in a more autonomous fashion. Further revision of the system is expected at this point.

Apart from these revisions certain features can be added to the system, depending on demand. These features include adding the learning algorithm mentioned before. The usefulness

of this feature for spacecraft is not very clear, but if use of the system were to branch out to robotics it would be an important addition. Another useful feature would be a port of the system to a massively parallel architecture, as mentioned before. The system seems well suited to such an implementation, and it would be both instructive and interesting to create such a port.

The final judge of the system will be the use it will be put to. The system is only as successful as the impact it has.

A.1 distream.h

```
#pragma once

#include "SharedSocket.h"

template<class T>
class distream {
// Default State:
//     Start from Begining
//     Input calls are blocking
//     eof condition will put thread to sleep
public:

       enum { AtCurrent=-1, FromBegining=-2 };

       distream(void);
       distream(string desc, int pos=FromBegining, bool wait=false);
       virtual ~distream(void);

       // desc: string to match to writers
       // pos: where to attach to the stream
       //     positive int(n) - attach n from begining
       //     distream::AtCurrent - wherever the stream currently is
       //     distream::FromBegining - from first data currently in the stream
       void open(string desc, int pos=FromBegining, bool wait=false);
       void close(void);

       distream &operator>>(T &data);

       bool eof(void);
       bool dataready(void);

       // Manipulators
       distream& operator>>(distream& (*f)(distream&)) { return f(*this); }
       distream& block(void) { blocking=true; return *this; }
       distream& nonblocking(void) { blocking=false; return *this; }
       distream& eof_on(void) { sleepOnEof=false; return *this; }
       distream& eof_off(void) { sleepOnEof=true; return *this; }
private:
       int myIndex;
       bool blocking, sleepOnEof;
       SharedSocket<T> *lpShSock;
       Lock lock;
};


// Manipulators
template<T>
distream<T>& block(distream<T>& di) {
       return di.block();
}

template<T>
distream<T>& nonblocking(distream<T>& di) {
       return di.nonblocking();
}
```

```
template<T>
distream<T>& eof_on(distream<T>& di) {
	return di.eof_on();
}

template<T>
distream<T>& eof_off(distream<T>& di) {
	return di.eof_off();
}

#include "distream.tmpl.h"
```

A.2 distream.tmpl.h

```
#include "StreamRegistry.h"

template<class T>
distream<T>::distream(void) :
		lpShSock(0), myIndex(-1), blocking(true), sleepOnEof(true) {
	if(!StreamRegistry<T>::lpTheRegistry)
		StreamRegistry<T>::lpTheRegistry = new(Heap::Shared())
StreamRegistry<T>;
}

template<class T>
distream<T>::distream(string desc, int pos, bool wait) {
	distream();
	open(desc, pos, wait);
}

template<class T>
distream<T>::~distream(void) {
	close();
}

template<class T>
void distream<T>::open(string desc, int pos, bool wait) {
	LockBlock<Lock> LB(lock);
	if(lpShSock) {
		lock.release();
		close();
		lock.acquire();
	}

	myIndex = StreamRegistry<T>::lpTheRegistry->RegisterReader(desc,
&lpShSock, pos, wait);
	DebugObj::Assert(lpShSock, "#distream::open - coulud not register
stream");
}

template<class T>
void distream<T>::close(void) {
	LockBlock<Lock> LB(lock);
	if(!lpShSock) return;

	StreamRegistry<T>::lpTheRegistry->CheckOutReader(lpShSock, myIndex);
	lpShSock = 0;
}
```

```
template<class T>
bool distream<T>::eof(void) {
	return !lpShSock || (!sleepOnEof && !lpShSock->dataready(myIndex));
}

template<class T>
distream<T> &distream<T>::operator>>(T &data) {
	DebugObj::Assert(!(myIndex<0), "#distream::operator>> - invalid stream
index");
	lpShSock->Get(data, myIndex, blocking, sleepOnEof);
	return *this;
}

template<class T>
bool distream<T>::dataready(void) {
	return lpShSock->dataready(myIndex);
}

template __dont_instantiate class distream<int>;
```

A.3 dostream.h

```
#pragma once

#include "SharedSocket.h"

template<class T>
class dostream {
public:
	dostream(void);
	dostream(string desc);
	virtual ~dostream(void);

	void open(string desc);
	void close(void);

	bool is_open(void);

	dostream &operator<<(T &data);

private:
	SharedSocket<T> *lpShSock;
	Lock lock;
};

#include "dostream.tmpl.h"
```

A.4 dostream.tmpl.h

```
#include "StreamRegistry.h"

template<class T>
dostream<T>::dostream(void) : lpShSock(0) {
	if(!StreamRegistry<T>::lpTheRegistry)
```

```
            StreamRegistry<T>::lpTheRegistry = new(Heap::Shared())
StreamRegistry<T>;
}

template<class T>
dostream<T>::dostream(string desc) {
       dostream();
       open(desc);
}

template<class T>
dostream<T>::~dostream(void) {
       close();
}

template<class T>
void dostream<T>::open(string desc) {
       LockBlock<Lock> LB(lock);
       if(lpShSock) {
              lock.release();
              close();
              lock.acquire();
       }

       StreamRegistry<T>::lpTheRegistry->RegisterWriter(desc, &lpShSock);
       DebugObj::Assert(lpShSock, "#dostream::open - could not register
stream");
}

template<class T>
void dostream<T>::close(void) {
       LockBlock<Lock> LB(lock);
       if(!lpShSock) return;

       StreamRegistry<T>::lpTheRegistry->CheckOutWriter(lpShSock);
       lpShSock = 0;
}

template<class T>
dostream<T> &dostream<T>::operator<<(T &data) {
       (*lpShSock) << data;
       return *this;
}

template<class T>
bool dostream<T>::is_open(void) {
       return (lpShSock!=0);
}

template __dont_instantiate class dostream<int>;
```

A.5 Instantiations.cp

```
#pragma once
#pragma export on

#define _COMMLIB

#include "dostream.h"

template class SharedSocket<int>;
```

```
template class StreamRegistry<int>;
template class dostream<int>;
template class distream<int>;

StreamRegistry<int> *StreamRegistry<int>::lpTheRegistry(0);
```

A.6 SharedSocket.h

```
#pragma once

/* #include "SystemHeap_allocator.tmpl.h"
#ifdef DefAllocator
#undef DefAllocator
#endif
#define DefAllocator SystemHeap_allocator
*/

#include <map>
#include <list>
#include "SleepList.h"

template<class T>
class SharedSocket {
public:

	enum { eofthrow=1 };

	SharedSocket(void);
	virtual ~SharedSocket(void);

	// Writer Functions
	void AddWriter(void);
	bool DelWriter(void);

	SharedSocket& operator<<(T &data);

	// Reader Functions
	int AddReader(int pos, bool wait);
	bool DelReader(int index);

	void Get(T& data, int index, bool blocking, bool sleepOnEof);

	bool dataready(int index);
private:
	class DataElem {
		public:
			// These guys inlined here because CodeWarrior craps out on
nested
			// classes when templated
			DataElem(void) : rcount(-1) {}
			DataElem(T &data, int nreaders) : elem(data),
rcount(nreaders) {}
			virtual ~DataElem(void) {}

			int rcount;
			T elem;
	};  // DataElem

	typedef list<DataElem> DataBuff;  // Add the whole SystemHeap_Allocator
stuff
```

```
        typedef DataBuff::iterator DataPtr;
        typedef map<int, DataPtr, less<int> > ClientMap;  // SystemHeap_Allocator
        typedef ClientMap::iterator ClientPtr;

        int nwriters, next_index;
        bool got_reader;
        DataBuff TheBuffer;
        ClientMap TheClients;
        SleepList slWaiting;
        Lock lock;
};

#include "SharedSocket.tmpl.h"
```

A.7 SharedSocket.tmpl.h

```
#include "SleepList.h"
#include "Threadmanager.h"
#include "distream.h"

template<class T>
SharedSocket<T>::SharedSocket(void) : nwriters(0), next_index(0),
got_reader(false) {}

template<class T>
SharedSocket<T>::~SharedSocket(void) {
        // Just make sure that everybody is done running.
        LockBlock<Lock> LB(lock);
}

template<class T>
void SharedSocket<T>::AddWriter(void) {
        LockBlock<Lock> LB(lock);
        nwriters++;
}

template<class T>
int SharedSocket<T>::AddReader(int pos, bool wait) {
        LockBlock<Lock> LB(lock);
        got_reader = true;
        while(wait && TheBuffer.size()==0)
                SleepOnList(slWaiting, lock);

        switch(pos) {
        case distream<T>::AtCurrent:
                TheClients[next_index] = TheBuffer.end();
                break;
        case distream<T>::FromBegining:
                TheClients[next_index] = TheBuffer.begin();
                break;
        default:
                TheClients[next_index] = TheBuffer.begin();
                for(int i=0; i<pos; i++) TheClients[next_index]++;
        }

        for(DataPtr d=TheClients[next_index];d!=TheBuffer.end();d++)
                (*d).rcount++;

        return next_index++;
};
```

```
template<class T>
bool SharedSocket<T>::DelWriter(void) {
	DebugObj::Assert(nwriters>0, "#SharedSocket::DelWriter - No Writer to
delete");
	LockBlock<Lock> LB(lock);
	return ((--nwriters)==0 && got_reader && TheClients.size()==0);
}

template<class T>
bool SharedSocket<T>::DelReader(int index) {
	DebugObj::Assert(TheClients.find(index)!=TheClients.end(),
"#SharedSocket::DelReader - No Reader to delete");
	LockBlock<Lock> LB(lock);
	for(DataPtr i = TheClients[index]; i!=TheBuffer.end(); i++)
		if((--((*i).rcount))==0) TheBuffer.erase(i);
	TheClients.erase(index);
	return (nwriters==0 && TheClients.size()==0);
}

template<class T>
SharedSocket<T>& SharedSocket<T>::operator<<(T &data) {
	LockBlock<Lock> LB(lock);
	TheBuffer.push_back(DataElem(data, TheClients.size()));
	for(ClientPtr i=TheClients.begin();i!=TheClients.end();i++)
		if((*i).second==TheBuffer.end()) {
			(*i).second = TheBuffer.end();
			(*i).second--;
		}

	slWaiting.Wake();
	return *this;
}

template<class T>
void SharedSocket<T>::Get(T &data, int index, bool blocking, bool sleepOnEof) {
	LockBlock<Lock> LB(lock);
	if(TheClients[index]==TheBuffer.end()) {  // Read when there is no more
data
		// If no more writers then if sleepOnEof go to sleep.  Else
(eof_on) throw eof
		// If more writers then if blocking go to sleep.  Else
(nonblocking) throw eof
		if((blocking && nwriters!=0) || (nwriters==0 && sleepOnEof)) {
			SleepOnList(slWaiting, lock);  // What if writers are done,
reader sleeps
		}                                                    // and then
all writers close?  Will never wake...
		else throw eofthrow;
	}

	DataPtr d = TheClients[index]++;
	data = (*d).elem;
	if((--(*d).rcount)==0) TheBuffer.erase(d);
}

template<class T>
bool SharedSocket<T>::dataready(int index) {
	return (TheClients[index]!=TheBuffer.end());
}

template __dont_instantiate class SharedSocket<int>;
```

A.8 StreamRegistry.h

```
#pragma once

/* #include "SystemHeap_allocator.tmpl.h"
#ifdef DefAllocator
#undef DefAllocator
#endif
#define DefAllocator SystemHeap_allocator
*/

#include <map>
#include <bstring.h>
#include "SharedSocket.h"

template<class T>
class StreamRegistry {
public:
	StreamRegistry(void);
	virtual ~StreamRegistry(void);

	void RegisterWriter(string desc, SharedSocket<T> **lplpSharedSock);
	int RegisterReader(string desc, SharedSocket<T> **lplpSharedSock, int
pos, bool wait);

	void CheckOutWriter(SharedSocket<T> *lpSharedSock);
	void CheckOutReader(SharedSocket<T> *lpSharedSock, int);

	static StreamRegistry<T>* lpTheRegistry;
private:
	typedef map<string, SharedSocket<T>*, less<string> > SockMap;
	typedef SockMap::iterator SockPtr;
	typedef SockMap::value_type SockVal;

	class IsEqual {
	public:
		IsEqual(SharedSocket<T> *s) : lpShSock(s) {};
		bool operator() (SockVal v) { return v.second==lpShSock; }
	private:
		SharedSocket<T> *lpShSock;
	};

	SockMap TheMap;
	Lock lock;
};

#include "StreamRegistry.tmpl.h"
```

A.9 StreamRegistry.tmpl.h

```
#include "Heap.nr.h"

#ifndef _COMMLIB

template<class T>
extern StreamRegistry<T> *StreamRegistry<T>::lpTheRegistry;

#endif // _COMMLIB
```

```
template<class T>
StreamRegistry<T>::StreamRegistry(void) : TheMap(){}

template<class T>
StreamRegistry<T>::~StreamRegistry(void) {
	// Just make sure that everybody is done running
	LockBlock<Lock> LP(lock);
}

template<class T>
void StreamRegistry<T>::RegisterWriter(string desc, SharedSocket<T>
**lplpSharedSock) {
	lock.acquire();
	if(TheMap.find(desc)==TheMap.end())
		TheMap[desc] = new(Heap::Shared()) SharedSocket<T>;
	lock.release();
	(*lplpSharedSock) = TheMap[desc];
	(*lplpSharedSock)->AddWriter();
}

template<class T>
int StreamRegistry<T>::RegisterReader(string desc, SharedSocket<T>
**lplpSharedSock, int pos, bool wait) {
	lock.acquire();
	if(TheMap.find(desc)==TheMap.end())
		TheMap[desc] = new(Heap::Shared()) SharedSocket<T>;
	lock.release();
	(*lplpSharedSock) = TheMap[desc];
	return (*lplpSharedSock)->AddReader(pos, wait);
}

template<class T>
void StreamRegistry<T>::CheckOutWriter(SharedSocket<T> *lpSharedSock) {
	bool DeleteFromMap = lpSharedSock->DelWriter();

	LockBlock<Lock> LB(lock);
	SockPtr s = find_if(TheMap.begin(), TheMap.end(), IsEqual(lpSharedSock));
	if(s!=TheMap.end() && DeleteFromMap) {
		delete (*s).second;
		TheMap.erase(s);

		if(TheMap.size()==0) {
			delete lpTheRegistry;
			lpTheRegistry=0;
		}
	}
}

template<class T>
void StreamRegistry<T>::CheckOutReader(SharedSocket<T> *lpSharedSock, int
index) {
	bool DeleteFromMap = lpSharedSock->DelReader(index);

	LockBlock<Lock> LB(lock);
	SockPtr s = find_if(TheMap.begin(), TheMap.end(), IsEqual(lpSharedSock));
	if(s !=TheMap.end() && DeleteFromMap) {
		delete (*s).second;
		TheMap.erase(s);

		if(TheMap.size()==0) {
			delete lpTheRegistry;
			lpTheRegistry=0;
```

```
            }
        }
}

template __dont_instantiate class StreamRegistry<int>;
```

Appendix B. Communications Demo

This is the source code for the demonstration of how to use the communications library.

B.1 SCDemoRxMain.cp

```
#include "ThreadStarter.h"
#include "SC Demo Rx thread.h"

void main( ) {
       NewRxThread NewRx;
       ThreadStarter *rxStart = new ThreadStarter(&NewRx, false);

       rxStart->Target("Receiver");
       (*rxStart)();
       delete rxStart;
}
```

B.2 SCDemoRxThread.cp

```
#include <iostream.h>
#include "SC Demo Rx thread.h"
#include "distream.h"

RxThread::RxThread (const sysstring &Name, Environment &E) : _FPThread
<Processor>(E) {}

void RxThread::Run (void) {
       try {
              int i;
              distream<int> InStream;
              InStream.open("Ints", distream<int>::FromBegining);

              InStream >> block >> eof_on;

              for(int n=0; n<10; n++) {
                     while(!InStream.eof()) {
                            InStream >> i;
                            cout << '<' << i << '>' << endl;
                     }
                     cout << "EOF!" << endl;
                     Delay();
              }

              cout << "Done Receiving" << endl;
       }
       catch(exception e) {
              cout << e.what() << endl;
       }
       catch (...) {
              cout << "Exception in RxThread::Run()" << endl;
```

```
        }
}

Thread* NewRxThread::operator() (const sysstring &Name, Environment *E) {
        return new (E->LocalHeap()) RxThread (Name, *E);
}
```

B.3 SCDemoRxTxMain.cp

```
#include "ThreadStarter.h"

#include "SC Demo Tx thread.h"
#include "SC Demo Rx thread.h"

void main( ) {
        NewTxThread NewTx;
        NewRxThread NewRx;

        ThreadStarter *txStart = new ThreadStarter(&NewTx, false);
        ThreadStarter *rxStart = new ThreadStarter(&NewRx, false);

        txStart->Target("Transmitter");
        rxStart->Target("Receiver");

        NewTx.start=0;
        (*rxStart)();
        (*txStart)();

        NewTx.start=10;
        (*txStart)();
        (*rxStart)();

        delete txStart;
        delete rxStart;
}
```

B.4 SCDemoTxMain.cp

```
#include "ThreadStarter.h"
#include "SC Demo Tx thread.h"

void main( ) {
        NewTxThread NewTx;
        ThreadStarter *txStart = new ThreadStarter(&NewTx, false);

        txStart->Target("Transmitter");
        NewTx.start=0;
        (*txStart)();
        delete txStart;
}
```

B.5 SCDemoTxThread.cp

```
#include <iostream.h>
#include "SC Demo Tx thread.h"
#include "dostream.h"

void TxThread::Run (void) {
	try {
		dostream<int> OutStream;
		OutStream.open ("Ints");

		for (int i = start; i < (start+10); i++)
			OutStream << i;

		OutStream.close ();

		cout << "Done Transmitting" << endl;
	}
	catch(exception e) {
		cout << e.what() << endl;
	}
	catch(...) {
		cout << "Exception in TxThread::Run()" << endl;
	}
}

Thread * NewTxThread::operator()(const sysstring &Name, Environment *E) {
	return new(E->LocalHeap()) TxThread(Name, *E, start);
}
```

B.6 SCDemoRxThread.h

```
#include "ThreadStarter.h"
#include "ThreadImpl.h"

class RxThread :  public _FPThread<Processor> {
	public:
		RxThread (const sysstring&, Environment&);
		void Init (void) {};
		void Run (void);
};

class NewRxThread : public _ThreadMaker {
	public:
		Thread* operator() (const sysstring& Name, Environment* E);
};
```

B.7 SCDemoRxThread.h

```
#include "ThreadStarter.h"
#include "ThreadImpl.h"

class TxThread : public _FPThread<Processor> {
	public:
		TxThread (const sysstring &Name, Environment &E, int n) :
start(n), _FPThread<Processor>(E) {};
		void Init () {};
		void Run (void);
		int start;
```

```
};

class NewTxThread : public _ThreadMaker  {
       public:
              Thread * operator()(const sysstring &Name, Environment *E);
              int start;
};
```

Appendix C. Commlib Differences

This is the file that was distributed with the new commlib explaining the difference from the previous release.

```
Differences from the old streams:
1) **Arguments to distream::open and distream::distream have changed**
2) **Manipulators have been added to configure the state of a distream**

3) Streams support multiple writers
4) You do not need to descend your threads from either pktsource or pktsink
5) You do not need to descend the data you want to pass from pktBase
6) There is no need to manually instantiate ANYTHING

1) The arguments to distream::open and distream::distream have changed, but
have defaults such that CodeWarrior will happily go ahead and let you compile
your old code.  Doing so will probably result in many calls to me asking what
the hell is happening.  The new arguments are:
       distream::open(string desc, int pos=FromBegining, bool wait=false);
       distream::distream(string desc, int pos=FromBegining, bool wait=false);
       desc - this is the string that provides the connection between distreams
              and dostreams.  Meaning is the same as in the previous library
       pos -  this determines where the distream will join up with the
              dostreams.
              FromBegining will position the stream at the beginning of the
                     available data.
              AtCurrent will position the stream at the end of the data buffer,
                     so that the first data it reads will be the first data
                     written to the stream after the reader's creation.
                     specifying a Positive Integer (n) will position the stream
                     n elements from the beginning of the available data.
              This defaults to FromBegining.

              *Note* - the streams only keep around data that some reader has
not yet read.  If every registered reader has read past some data, that data is
released from the buffer.

wait - this boolean controls how the stream behaves if there is no data
available     when the stream is opened. If wait is true, the stream will put
the thread to sleep and will wake up when the first data gets written. If wait
is false, the stream will register itself and return. This defaults to false.

2) The following manipulators have been defined for distreams: block,
nonblocking, eof_on, eof_on.  They affect the way the stream behaves when it
attempts a read when there is no data available.

       stream << block << eof_on - In this configuration, if a distream reads
when there is no more data available, one of two things happens.  If there are
still writers registered, the stream will put the thread to sleep until more
data is available.  If no more readers are available, then the call will throw
an  exception.

       stream << block << eof_off - This configuration is similar to the above
except when there are no more writers.  In this case the stream will put the
thread to sleep and will wake up when there is more data (rather than throwing
an exception).
```

stream << nonblocking << eof_on - In this configuration, the stream will never put the thread to sleep. If you read beyond the eof at any time, an exception will be thrown.

stream << nonblocking << eof_off - In this configuration, the stream will only put the thread to sleep when it tries a read when there are no writers and there is no data available. If you try to read past the eof, the call will throw an exception. Since eof has been turned off (i.e. calls to eof() will always return false), this mode only makes sense when you are sure that the reader will never read faster than writer will write.

3) There is no special syntax for attaching multiple writers, simply declare each writer like you normally would. The writes are done asynchronously, so the data from different threads may be interleaved.

As a matter of fact, I'm not even going to provide the pktsource and pktsink base class. If you don't want to release your threads from this yoke, include the files from the previous streaming library.

5) See 4. Replace pktsink and pktsource with pktBase.

6) Just declare streams around any class that has a copy constructor. CodeWarrior can figure out the correct instantiations to make.

** Note that these streams cannot communicate across memory contexts. Support for this is the next thing to be added.

Bibliography

C. M. Angle. Genghis, a Six Legged Walking Robot. Masters thesis, MIT. (1989)

E. Bizzi, F. Mussa-Ivaldi, S. Giszter. Computations Underlying the Execution of Movement: A Biological Perspective. Science 253, 287-291. (1991)

R. Bonasso, R. Firby, E. Gat, D. Kortenkamp, D. Miller, M. Slack. Experiences with an Architecture for Intelligent, Reactive Agents. Journal of Experimental and Theoretical Artificial Intelligence. (1997)

R. Brooks. A Robot That Walks: Emergent Behavior from a Carefully Evolved Network. Neural Computations 1(2), 253-262. (1989)

R. Brooks. Intelligence Without Representation. Artificial Intelligence Journal 47, 139-160. (1991)

P. Dana. Global Positioning System Overview. http://wwwhost.cc.utexas.edu/ftp/pub/grg/gcraft/notes/gps/gps.html. (1994)

C. Dunn. GOAC Firmware Description & Status. Internal Rogue Group Memo. (1997)

C. Dunn. GOAC Firmware API Overview. Internal Rogue Group Memo. (1998)

F. Fisher, S. Chien, L. Paal, E. Law, N Golshan, M. Stockett. An Automated Deep Space Communications Station. IEEE Aerospace Conference. (1998)

G. Franklin. Rogue Group GPS Receiver Development. http://rogueweb.jpl.nasa.gov/. (1998)

B. Hayes-Roth. An Architecture for Adaptive Intelligent Systems. Artificial Intelligence, 72. (1995)

S. Kirby. Language evolution without natural selection: From vocabulary to syntax in a population of learners. Edinburgh Occasional Paper in Linguistics (1). (1998)

J. Lettvin, H. Maturana, W. McCulloch, W. H. Pitts. What the Frog's Eye Tells The Frog's Brain. Proceedings of the IRE, Vol. 47, No. 11, 1940-51. (1959)

T. Lozano-Perez. A Simple Motion Planning Algorithm for General Robot Manipulators. IEEE Journal of Robotics and Automation, Vol RA-3, No. 3, 224-238. (1987)

M. Mataric. Integration of Representation Into Goal-Driven Behavior-Based Robots. IEEE Journal of Robotics and Automation 8(3), 304-312. (1992)

M. Minsky. The Society of Mind. Touchstone Books, published by Simon & Schuster. (1985)

N. Muscettola, C. Fry, K. Rajan, B. Smith, S. Chien, G. Rabideau, D. Yan. On-Board Planning for New Millennium Deep Space One Autonomy. Proceedings of the IEEE Aerospace Conference. (1997)

N. Nilsson. Shakey the Robot. Stanford Research Institute AI Center, Technical Note. (1984)

B. Pell, E. Gat, R. Keesing, N. Muscettola, B. Smith. Plan Execution for Autonomous Spacecraft. Proceedings of the AAAI Fall Symposium on Plan Execution. (1996)

S. Rao. Visual Routines and Attention. PhD thesis, MIT. (1998)

S. Shekhar, B. Hamidzadeh. DynoraII: A Real-Time Planning Algorithm. Intl. Jr. on Artificial Intelligence Tools 2(1), 93-115. (1993)

R. Simmons. Structured Control for Autonomous Robots. IEEE Transactions on Robotics and Automation, 10(1). (1994)

B. Smith, K. Rajan, N. Muscettola. Knowledge Acquisition for the Onboard Planner of an Autonomous Spacecraft. Proceedings of the European Knowledge Acquisition Workshop. (1997)

Trimble Navigation Limited. What is GPS? http://www.trimble.com/gps/index.htm. (1996)

S. Ullman. High Level Vision: Object Recognition and Visual Cognition. MIT Press. (1996)

W. G. Walter. The Living Brain. Pelican Books. (1953, republished 1961)

P. H. Winston. Learning Structural Descriptions From Examples. PhD thesis, MIT. (1970)

K. Yip, G. Sussman. A Computational Model for the Acquisition and Use of Phonological Knowledge. MIT Artificial Intelligence Memo No. 1575. (1996)